UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Point Cloud Data Fusion for Real-Time Applications

Author:
Stéphane Maillot

Supervisor/s:
Joan Aranda Lopez
Manuel Vinagre Ruiz

A thesis submitted in partial fulfillment for the
Master's degree in
Automatic Control and Robotics

in the
Barcelona School of Industrial Engineering
Department of Systems Engineering,
Automation and Industrial Computing

September 2018

Abstract

This thesis presents how to register point clouds from RGB-D sensors to perform real-time 3D reconstruction of an indoor scene with the aim to build a representation of the environment useful for robot planning.

Several common techniques have been explored and compared to determine benefits and drawbacks of each method for our setup.

A more specific method was then developed based on this knowledge and has been compared with other methods in terms of rotation and translation errors.

Finally these results are used to evaluate the method accuracy and determine in which conditions these results can be replicated.

Contents

Preface

Origin of the project

Although robots have been used for decades in the industry, one of the current challenges for robotics is to adapt robots for domestic environment to help people in their everyday life. Thus, this project aims to teach robots how to interact with humans in a kitchen environment in order to help them achieving common actions in a collaborative way. This project particularly applies to disabled people who may need assistance for achieving daily tasks.

Motivation

To achieve this goal, we need to acquire a detailed representation of the scene before being able to act on the environment with robots. Using computer vision to build a detailed and real-time 3D map of the scene in real time is then a necessary step to achieve in order for the robots to understand their environment before choosing a desired action and planning a trajectory to actually perform this task. This project uses the Kinect V2 depth sensors to acquire 3D data from the scene. Kinect sensors have a better accuracy at close range and then, even when it is feasible, it will not be the best idea to put the sensor far away from the workspace in order to acquire data from the whole environment with only one sensor.

Thus, it is preferable to reconstruct the scene from partial views of the environment to gain accuracy when the scene is too large to use only one close range camera. It is then needed to merge different views into one single complete representation of the workspace. The main motivation of this work is then to be able to

reconstruct the complete 3D map in order for the robots to be able to understand their environment.

Requirements

This work mainly requires depth sensors, we are using Kinect V2. These sensors provide an high frequency (default: 30 FPS) stream of RGB and depth data which are converted into 3D colored Point Cloud. The computer needed to process this amount of data have to be powerful if we want to process it with high frequency. In this project we use a Titan X GPU and i7 cores to process the data.

For the implementation, this work is entirely made using free open source software. It is running on Linux OS using ROS middleware and open source libraries (PCL, openCV and other ROS libraries).

Chapter 1

Introduction

1.1 Objectives

The aim of this work is, given 2 point clouds from different kinect sensors, to merge them into a new point cloud containing all the information needed to build the 3D map of the workspace. To achieve this goal we already have some well know techniques to calibrate the sensors.

Until now, this project was using a registration process involving printed markers recognition. However, it appears that this process has some disadvantages. First, it requires a special printed pattern that can be recognized by a program to estimate camera positions. This manipulation requires to be done each time a sensor is moved. Intentional movements of the cameras (when changing the workspace setup) may not be frequent but unexpected small rotations of the kinects can occur more frequently. For example, a camera rotation of 1° will produce several cm of error on distant points. Along time, this will lead to big errors in scene reconstruction and the robots may not be provided a 3D map accurate enough to behave as expected.

The purpose of this work is then to find a calibration method that fits better our requirements. In this section I will detail some of the main requirements of this work that I need to focus on to make sure that the end result will be useful for the other subsystems.

1.1.1 Spatial accuracy

In the first place, we need the result of this reconstruction process to be accurate; distance between any point provided in the point cloud and its real position in the real scene should be small. In this project, the output point cloud will be used to build a voxelmap. This representation is using small cubes with fixed size to represent the scene. In the ideal case, the output point cloud contains points which position has a smaller error than this voxel size. In this case the map would represent accurately the position of each significant obstacle in the workspace as if it were computed from only one camera with a large view range. The common voxel size we use in this project is 2cm but it may vary depending on the task. However, the whole robot planning process couldn't rely entirely on this voxelmap which will mainly be used to segment planes and object to make decisions and create a rough trajectory planning that can be refined with more accurate embedded short range sensors.

1.1.2 Processing speed

The current process using marker board is now needing few minutes to provide accurate results. We want the new algorithm to run, in the worst case within the same amount of time which will allow us to use it as a calibration program. However, if this program is running faster, it could be called during the reconstruction process to recalibrate more frequently to make sure that no unexpected movement of the cameras has occurred. If fast enough, it could even be run in real time to estimate camera position in each frame which would be useful for tracking an embedded camera, installed on a moving robot for example.

1.1.3 Simplicity

The main aspect of this project is to evaluate how much we can automatize the registration process so that it doesn't involve any complicated process such as printing a template and moving it into the workspace for few minutes. The goal being to have a fully-automated process that doesn't requires any user action.

1.2 Challenges

In this particular project, we have to deal with unusual difficulties. Indeed, unlike most papers focusing on point cloud registration theory, we work with real noisy data from depth sensors instead of clean point clouds extracted from 3D models. Some papers add noise to their point clouds or use real point clouds (often laser scans that are much more precise) to test noise robustness of their algorithm. However we have to deal with other limitation from the kinects, even after careful intrinsic calibration, it remains image deformation, specially in the corners of the image. Differences in IR light reflection depending on the materials also deteriorate the quality of the point cloud by deforming or loosing some parts of the cloud. In addition we are not able to get a large overlapping region between the 2 point clouds for two reasons. The first is that, we want to cover a large scene and placing the kinects too close would reduce significantly the range of our system. The second reason is that kinects are active sensors that interfere within each other. This sensor use an IR projector to evaluate the travelling time of light rays, if another sensor is watching the same region, IR projections will interfere which leads to a nosier point cloud.

All of these limitations can be solved separately, but dealing with all of them in the same registration process requires to adapt some of the existing algorithms found in the literature.

1.3 Scope

This work is applied in a really particular project but the resulting technique could be applied in other cases as the aim of the project is not to write a solution to this particular scenario but to explore the possibility to reconstruct a scene with similar difficulties with as less knowledge as possible. My main assumption is that the scene is an indoor environment (kinect sensors can't be use outdoor) which gives us some clues about the geometry of the scene. Indoor environments are most likely to contains several big planes (walls, table or any flat furniture...). These planes provides reliable knowledge on camera orientation and assuming to be indoor will mainly allow me to be confident that we are able to detect planes in the input point clouds.

Also, I assume in this work that both camera intrinsic calibration is already solved, which means that, apart from calibration error, both clouds have similar scales (distances are the same within both point clouds) and non-deformed (angles are the same in both point clouds).

In this work, I will first present some of the most used techniques in 3D point cloud registration defining their field of application as well as their assets and drawbacks. I will then describe how we can overcome the previously mentioned difficulties by combining some of these state-of-the-art techniques. The next chapter will detail about the actual implementation of this solution and the final one will discuss the results and compare with other algorithms.

Chapter 2

State of the Art

2.1 Marker detection

The registration technique currently used in this project is the detection of AR marker in the color image. AR markers are black and white grid-like patterns (fig. 2.1) developed for augmented reality usage to detect easily a position in space from an RGB camera.

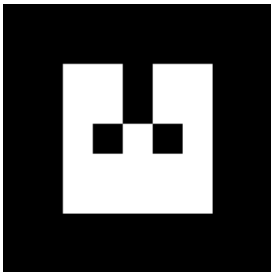

FIGURE 2.1: AR Marker.

This detection is not very accurate but by repeating the process hundreds of time and averaging we can get a good estimation of the position of the cameras. An obvious drawback regarding our needs for this project is the involvement of the user and the equipment requirement. The user needs to place a printed pattern into the overlapping region and no other work can be done while performing the registration.

To perform this registration, we place the marker in the overlapping region so that both cameras can be calibrated from this marker (fig. 2.2). As we have already discussed, the overlapping region cannot be trusted due to its poor quality

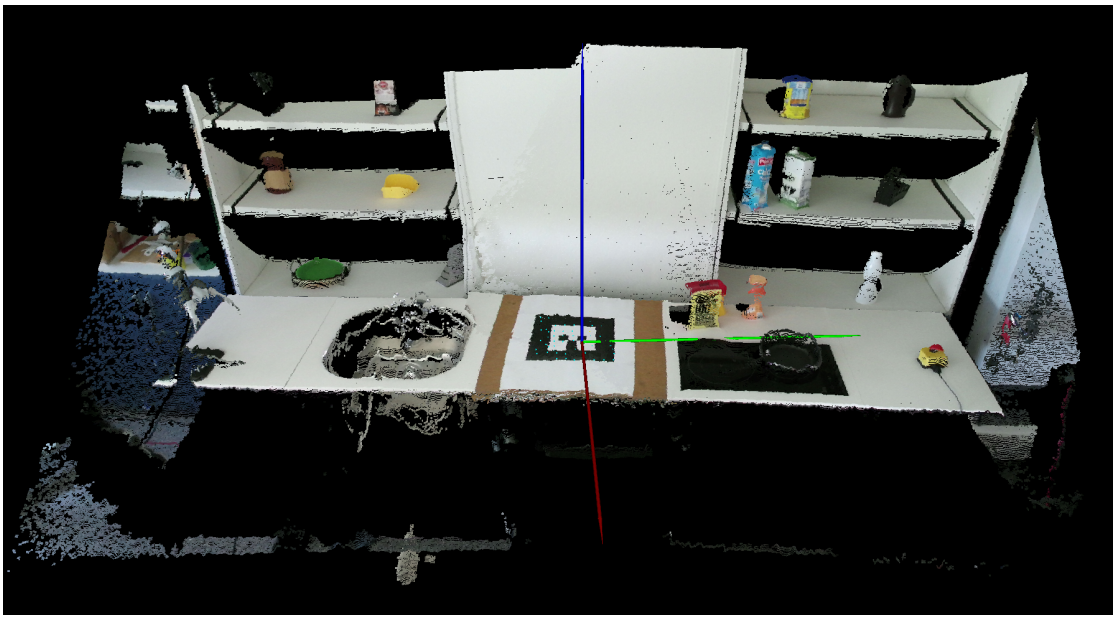

FIGURE 2.2: AR Marker detection.

and small size. This can be solved by calibrating both cameras with a different marker position (in the middle of each camera view), these positions have now to be know if we want to match point clouds together. This is a high constraint both for the user to perform the calibration twice and the workspace to be designed such that we can easily move the marker in it while still knowing precisely the position of the marker because any uncertainty on this measurement will be added to the registration error. This is the reason we are searching for a faster and more automatic registration process for this project.

For the implementation, we are using an existing ROS package based on alvar, which tracks AR printed patterns in a color image to estimate its position with respect to this camera. By filming a marker which position in the scene is known with a fix camera, we can average the pose estimation on hundreds of frames to get a good estimation of the actual position of the camera.

The main drawbacks of this technique are the need of printed pattern, human action, time of computation and knowledge of the scene. This is still a good technique to establish our ground truth and to build the entire scene point cloud we could need for other methods such as ICP.

2.2 Iterative Closest Point

The Iterative Closest Point (ICP) algorithm is a well-known registration algorithm that aims to register 2 point clouds that represent the same scene using an iterative process. These iterations rely on identifying closest points from both point clouds at each step in order to estimate a transformation that will improve the clouds matching without taking account of distant points (considered as outliers).

The first step of this algorithm is to find matches between points from the source point cloud $\mathcal{P}_\mathcal{S}$ to points in the target cloud $\mathcal{P}_\mathcal{T}$. As we have no knowledge of which point within the target cloud should match with source points, the key idea is to match source points with their closest neighbors in the target cloud (fig. 2.3). After filtering duplicate matches we have pairs of matching points with indices m_S and m_T. At this step we define the objective function f which is the Mean Square Error (MSE) of these matches according to the current guess for the rigid transform $\mathcal{T}_i$.

$$f(\mathcal{T}_i) = MSE(\mathcal{P}_T^{m_T} - \mathcal{T}_i\, \mathcal{P}_S^{m_S})$$

Where the initial guess $\mathcal{T}_0$ should be provided to the program, the quality of this guess will affect the final result.
After that we can compute the transform that minimize f for this set of matches. Having a new estimation of the rigid transform, we can repeat the process until convergence.

The main drawback of this technique come from the need of an initial guess, the ICP algorithm is used for refinement once we have a rough idea of the desired result. Thus, this method is not appropriate when we have no knowledge on camera relative position and orientation. In our case for example, the first calibration is done without having any knowledge and will fail. However it could still help to fix small calibration error that occurs along time. However the low overlapping ratio of our point clouds makes this techniques unusable, especially because the overlapping region is mostly flat, noisy and deformed (fig 2.4). A method based on point matching in these regions can't obtain a good result in our setup and we most likely obtain a bad registration as sketched on fig 2.5. We can see on this figure that the metric we use (closest points distance) is not appropriate to solve this problem since the expected result has a larger value than our result.

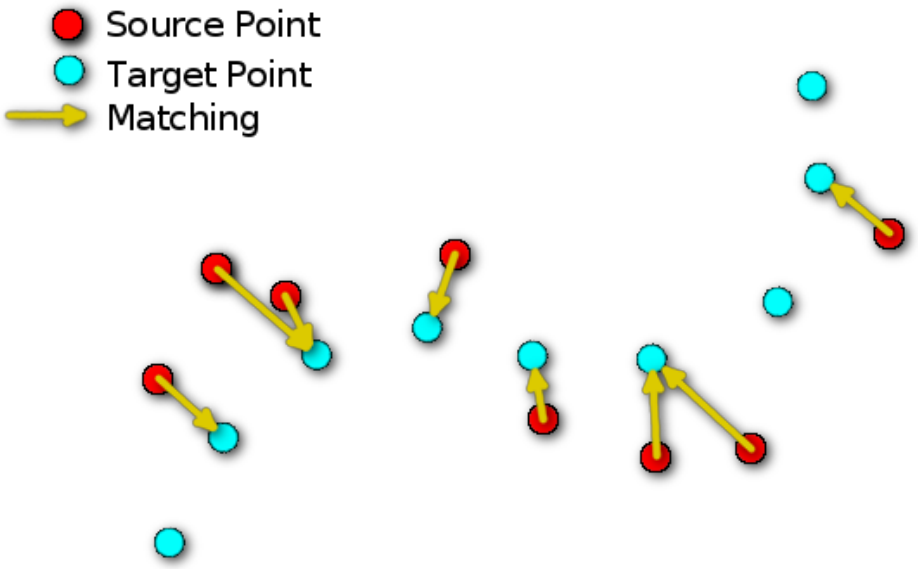

FIGURE 2.3: ICP closest point matching.

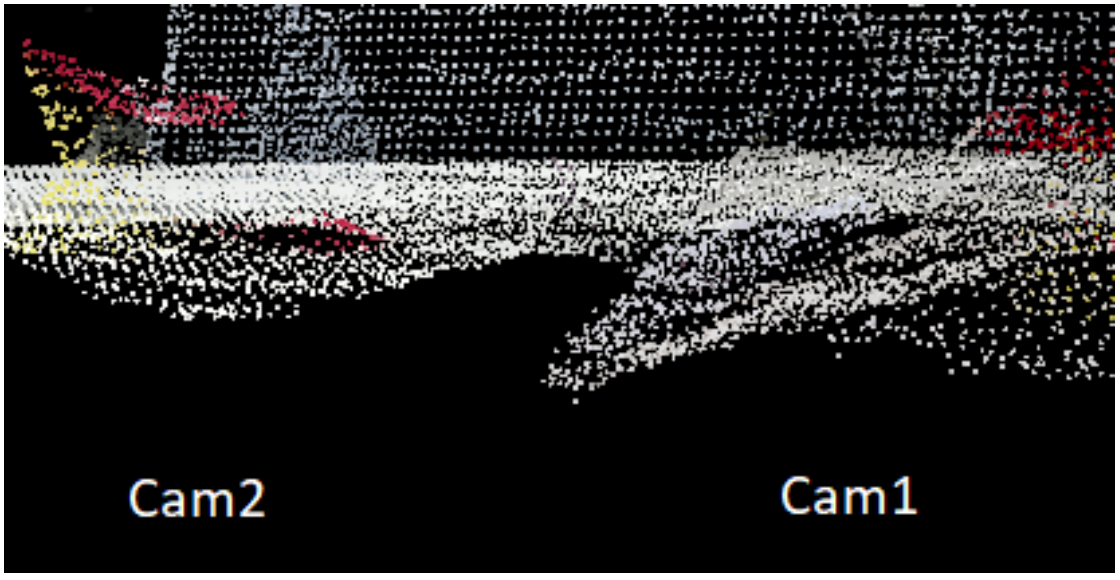

FIGURE 2.4: Overlapping region in both point clouds. We can observe that this region is really deformed which affects a lot the performance of this method. Camera relative position is set arbitrarily here for visualization purpose.

This algorithm also requires a huge amount of computation power if running directly on kinect point clouds which contains more than 16 million points. To run this algorithm faster we have, apart from decreasing the number of iteration and the convergence threshold, to downsample point clouds before running the algorithm. It may decrease the precision of the transform estimation but has no

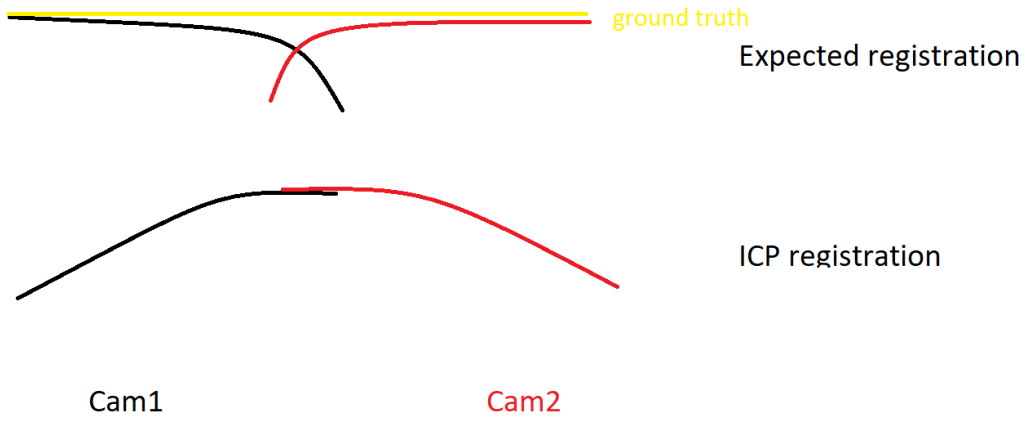

FIGURE 2.5: Impact of bad quality point cloud on the registration result. For example the table plane (yellow line) which suffers a big deformation in the overlapping region on both cameras, has shown in 2.4.

incidence on the final point cloud density since I can simply use the downsampled cloud to estimate the transform and then apply it to the full point cloud.

To perform registration using the ICP method, I am using the PCL implementation of this algorithm. As expected, applying this algorithm with our 2 low overlapping point clouds ends up to really bad results (fig. 2.6), however this technique is more effective when registering a partial view of the scene with a complete model of it (fig. 2.6). I was then able to apply ICP to both point clouds to register them to a more complete 3D model of the scene. This works both when applied to registration with a CAD model of the kitchen and to a complete point cloud generated by merging both kinect views but even with this model we don't have enough information to fix accurately the Y translation (green axis in fig. 2.1).

This algorithm is then useful when point clouds have a large overlapping or if we have previously built a model of the scene. As we are not assuming any knowledge on the environment (else than being indoor) and due to the low overlapping of our point clouds, this technique will not be used for the first registration of kinects. This can remain a good registration method as soon as we have created a complete and reliable model of the scene as shown in fig. 2.7. The scene model is provided by the simulated environment discussed in 4.1.2.

FIGURE 2.6: ICP applied to register one camera with the other.

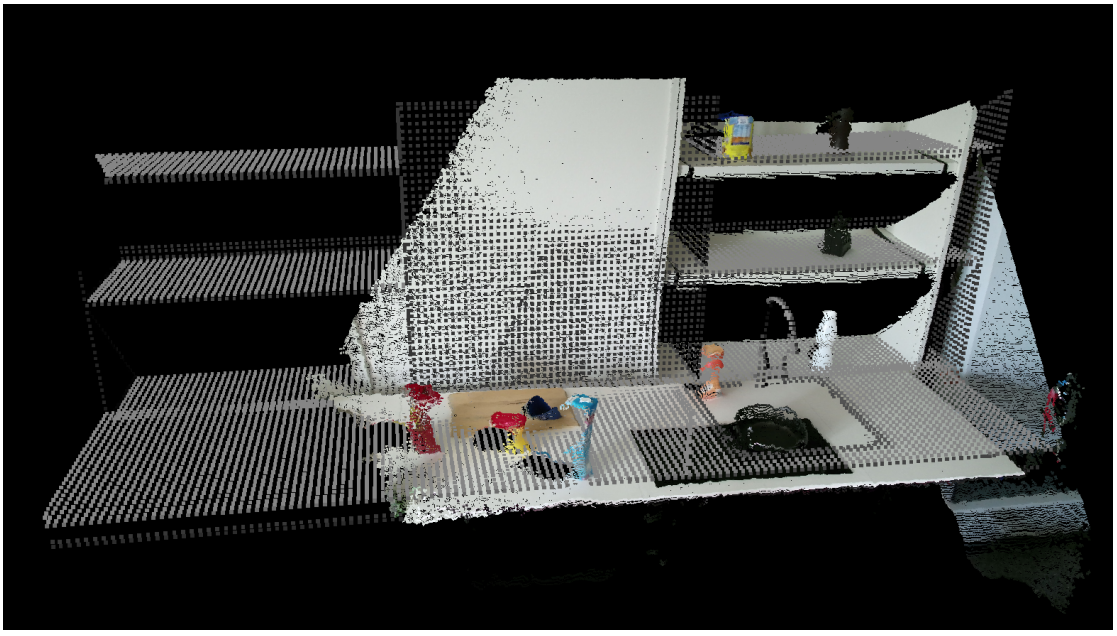

FIGURE 2.7: ICP applied to one camera and a 3D model of the kitchen (without objects).

2.3 3D Keypoints

Another commonly used registration technique consists in 3D keypoints detection and matching. There are a lot of different ways to perform registration using 3D points matching, but the process pipeline is usually similar as the following:

2.3.1 Keypoints extraction

Input point clouds contains way too much points to run the entire process on all of them. It is thus required to select only a subset of them, we call these selected points, keypoints. As explained in [Tom13], we can use several different keypoints extraction techniques such as ISS, NARF, SIFT or uniform sampling.

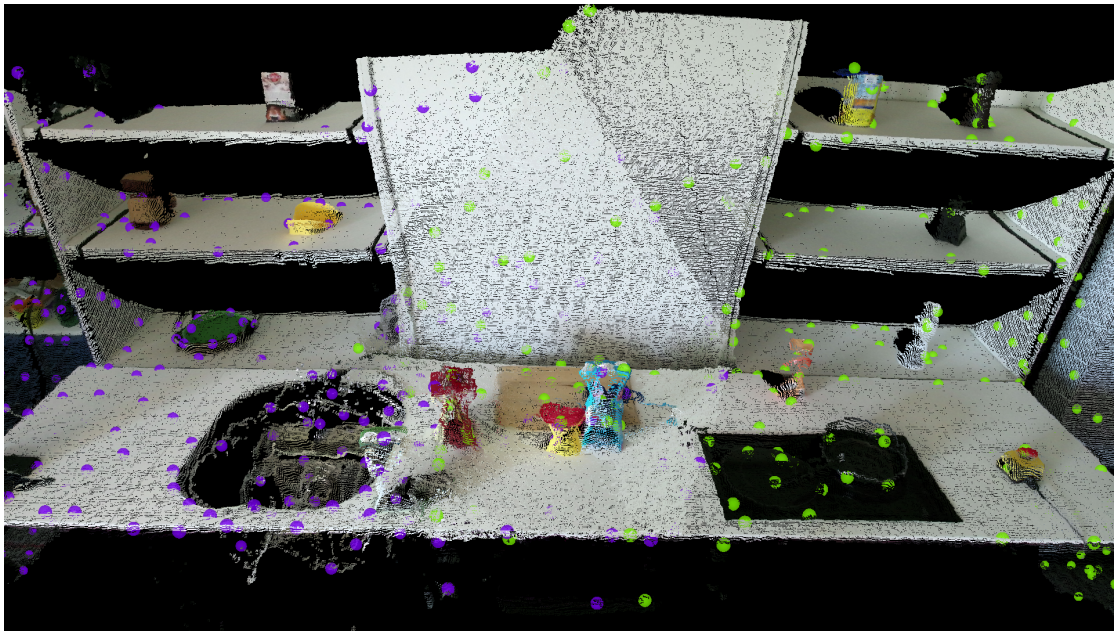

FIGURE 2.8: ISS keypoints extracted from both cameras.

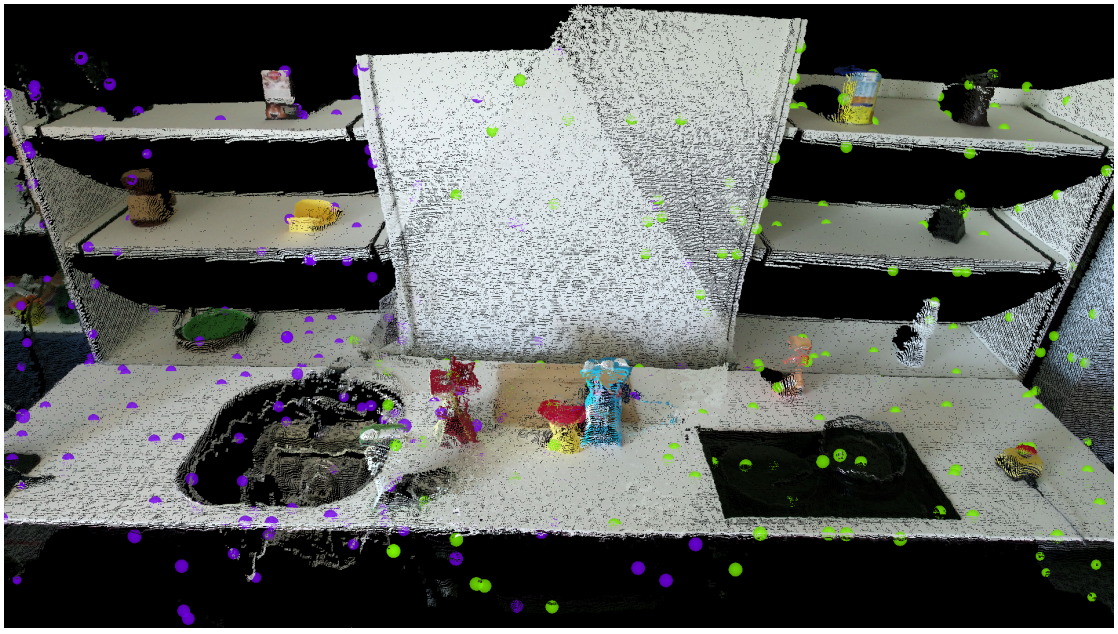

FIGURE 2.9: SIFT keypoints extracted from both cameras.

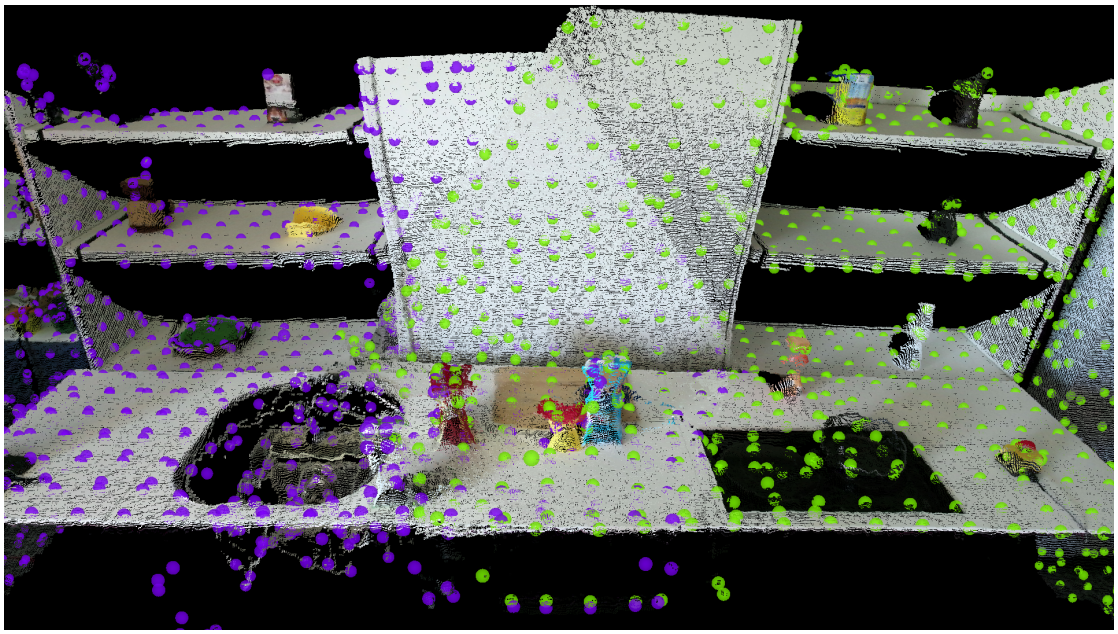

FIGURE 2.10: Uniform sampling of both cameras.

I obtained better results and fast computation using ISS and used only this one in later works.

2.3.2 Features extraction

There are a lot of different features we can extract from keypoints but it generally consists in stacking 3D shape and/or color histograms into a vector descriptor, if this descriptor is local, it can then be used to match the most similar keypoints to identify similar points (i.e. that represent the same real point in space) from both clouds. This matching process means that this method is also relying only on overlapping regions to perform registration.

For each keypoint, we want to be able to find the most similar keypoint in the second cloud, so it is needed to describe the point surrounding in a way that is invariant to rotation (its equivalent point in the other cloud is probably oriented differently unless both clouds are yet aligned) and translation (scale invariant is not a property we are interested in here since we assume cameras are intrinsically calibrated). This description is usually based on geometric features (relative position or density of its neighbors) and/or color features (color distribution around the point).
Due to the poor quality of the point cloud, especially in the region where we are searching for keypoints matching, I chose to use the CSHOT estimator from PCL

library that is extracting features from color information as well as 3D shape as explained in [TSS11].

2.3.3 Matching

These descriptors can now be matched by finding most similar points by comparing their features. For each point of the source point cloud we must measure the distance with every point in the target cloud to find the closest.

This process can then be really slow, to fasten this search, it is common to use KD-Trees. Given one set of features computed for every points in one of our clouds, we are building a tree such that each node divide the tree in 2 sub-trees in which every points' coordinate (in the feature space) is whether bigger or smaller than the median value for one particular axis. This process is repeated successively for every axis until every point is represented by a singular node or leaf. This algorithm can be written recursively as detailed in alg. 1.

Algorithm 1 KD-Tree

L is a list of feature vectors of size k

function KDTREE($L, depth$)
  $axis \longleftarrow depth \mod k$
  $m \longleftarrow$ median of L in axis direction
  create *node*
  $node.location \longleftarrow m$
  $node.leftchild \longleftarrow KDTREE(l \in L \mid l_{axis} < m,\ depth + 1)$
  $node.rightchild \longleftarrow KDTREE(l \in L \mid l_{axis} \geq m,\ depth + 1)$
  return *node*

2.3.4 Outliers Rejection

It is then often required to refine the previous matching by eliminating undesired matches such as if several points are matching with the same target point (we'll then keep only the best match, i.e. with the smallest descriptor distance), called duplicates, points that are too far (in space) according to the initial guess or points with descriptor distance over a given threshold.

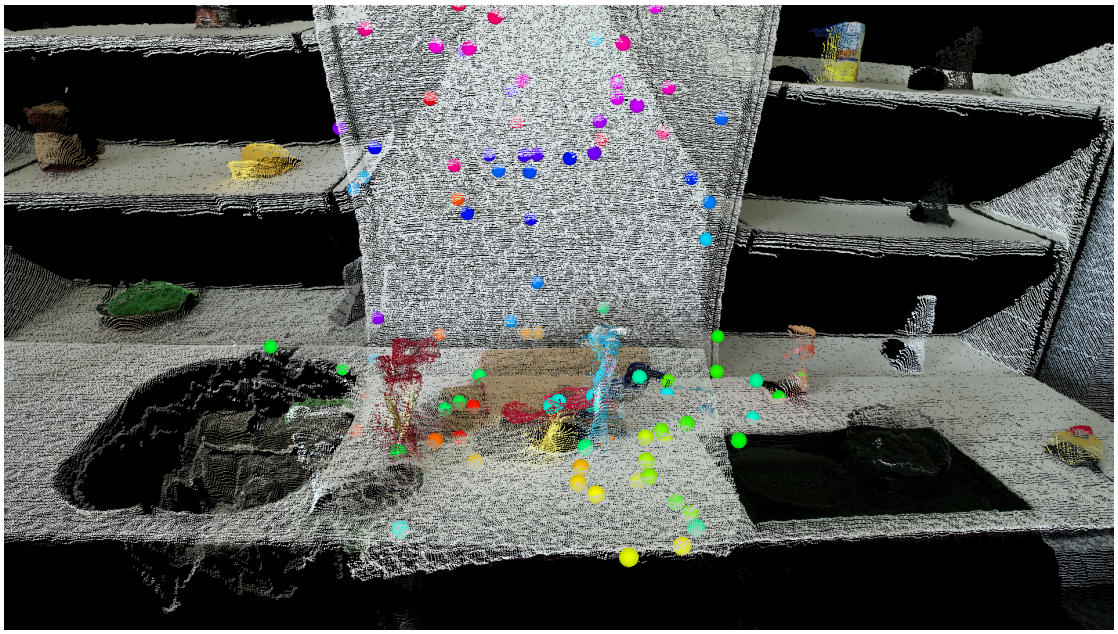

FIGURE 2.11: ISS keypoints matched between both views (same color = match).

2.3.5 Transform Estimation

The last step is then to estimate the transform that brings source matched points to their corresponding target points. Let p_i^s, p_i^t be the i^{th} matching pair of points (whose coordinates are expressed with respect to their centroids p'^s and p'^t) from source and target point clouds. Stacking N matching points into $3 \times N$ matrices $P^s = (p_0^s, p_1^s, \cdots)$ and P^t, we can define $\mathcal{T}$ as the solution of $\mathcal{T}P^s = P^t$. As no exact solution can be founds with real data, we want to find the transform $\mathcal{T}$ minimizing the quadratic error $\sum_i |p_i^t - \mathcal{T}p_i^s|^2$.

We can find separately the rotation and translation parts of the transform R, t such that $\mathcal{T} = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}$. The least-square solution can be derived as detailed in [SHR17], rotation part is given by:

$$R = U \begin{pmatrix} 1 & & \\ & 1 & \\ & & det(UV^\top)) \end{pmatrix} V^\top$$

Where U and V are computed from the SVD of the 3×3 correlation matrix K:

$$K = UDV^\top = \sum_i p_i {p_i^s}^\top$$

Then t can simply be computed as the vector:

$$t = p'^{t} - Rp'^{s}$$

2.4 Plane detection

An other interesting method that can be found in some papers is based on plane detection and matching. This method can solve the issue caused by low overlapping. Indeed, planes can be used to estimate how to merge point clouds such that planes from the , but the interesting property is that the plane equation can be computed with every point that lies into the plane model whether it is in the overlapping region or not. Thus, if the cloud contains large plains that can be identified with a large number of points spread in a large part of the point cloud, the confidence on this plane's equation will be far better than if we try to identify features in the overlapping region.

However, to determine the whole 6 degrees of freedom of the transformation between camera positions, we would need 3 non-degenerate planes (i.e. which normals describe a basis, the more orthogonal the better) to match in both point clouds. This condition is often too constraining to be satisfied, or is satisfied only if considering small and/or almost degenerate sets of planes. In this case, the accuracy gain provided by the plane identification is lost and one degree of freedom is left with a lot of uncertainty. Nevertheless, in most cases we can at least fix quite accurately the rotation between clouds (2 planes needed).

To apply this technique, I have used RANSAC algorithm in order to achieve plane detection (fig. 2.12).

This algorithm is detailed in alg. 2 in the case of plane detection, we have to repeatedly pick random points (3 to identify a plane model) and build the corresponding model to count how many points are lying to this model with a given threshold. This process is iterated a given number of times, the best model (with the largest number of inliers) is then returned as the more accurate estimation for this set of points.

Algorithm 2 RANSAC

cloud is the point cloud data
n the number of iteration needed
ε is the inlying threshold

```
1: function RANSAC(cloud, k, ε)
2:     n_max = 0
3:     FOR i from 0 to k
4:         randomly choose 3 points from cloud
5:         Build the plane model P containing these points
6:         n ⟵ number of points in cloud which distance to P is smaller than ε
7:         IF n > n_max
8:             n_max ⟵ n
9:             P_best ⟵ P
10:    return P_best
```

Similarly to the previous registration method, the next step consists in matching extracted planes.

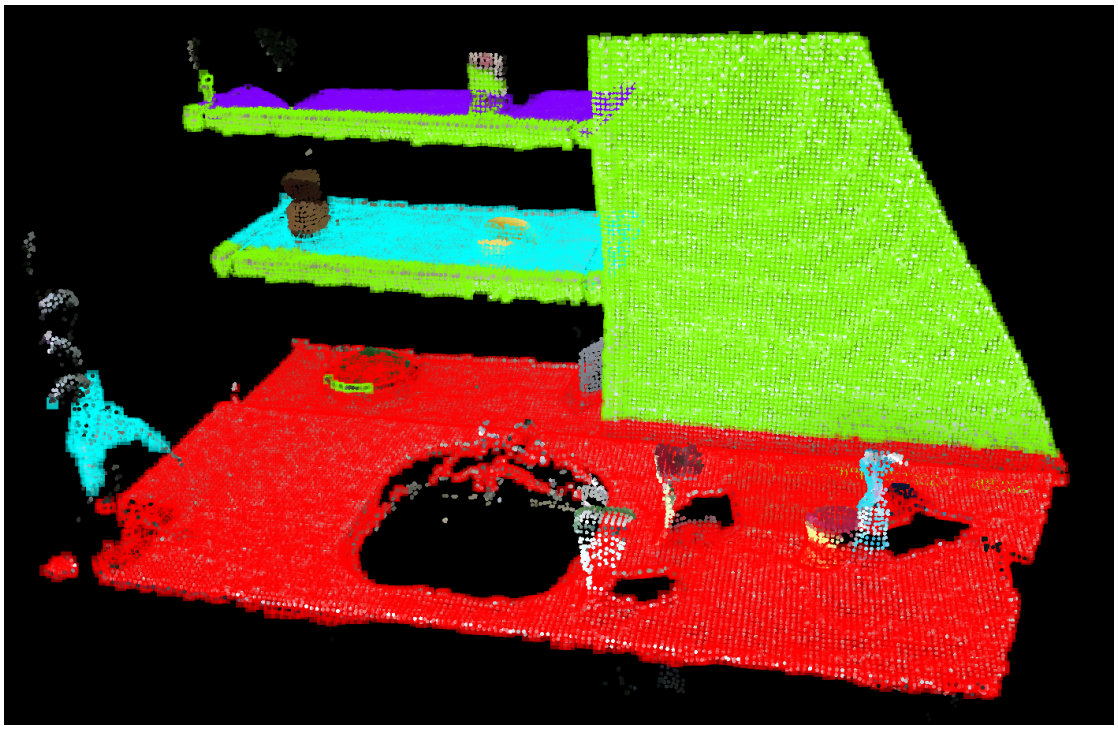

FIGURE 2.12: Plane extraction on one camera, $n = 4$.

My first try was to match using feature matching in the same way as I did with keypoints descriptors but using global descriptors (descriptors that contains information on the entire plane). The descriptors I tried to use for this matching are VFH global descriptor from PCL library and color histogram as sugested in

[DGFF13]. Further explanations are given in sec. 3.3. However, due to the similarity of planes in this particular scene (here, every planes is detected as an almost entirely white rectangle which fails the matching algorithm most of the time). This technique is then not very reliable in our case because of planes similarity and the lack of overlapping. Indeed, having a large overlapping would allow us to detect more specific patterns or textures common to both planes, here even the patterns we may recognize (sink, heating plate) are not seen from both cameras and can't be used for matching.

Another technique could simply be to use brute force matching, since we are usually not trying to detect a lot of planes (we need 3 of them), by trying every possible matching we could make sure to keep the one that ends with the best registration result. Nevertheless this technique should only be used after optimizing the computation speed and ensuring that we can afford to multiply processing time by 6 (at least, when we want to detect only 3 planes) since for each possible match we have to perform the complete registration process and then conclude which of the 6 results is more likely to be correct.

A simpler but still efficient way I implemented this matching was using planes normals and matching planes with the more similar orientation (details are given in sec. 3.3). This way, as long as both point clouds are not built from 2 sensors with completely different orientation, the matching will be done correctly. To avoid wrong matching when cameras are flipped, a previous step to find a rough estimation of the transform may be needed. Matching result can be seen in fig. 2.13.

The last step is to estimate the rigid transform that minimize plane-to-plane matching error as suggested in [Kho16].
A plane is defined as a 4D vector $\pi = (n^\top, -\rho)$ (normal vector and distance to origin) that satisfy for every point $p = (x, y, z, 1)$ of this plane the equation:

$$\pi p = n \cdot p_{xyz} - \rho = 0$$

If H is the transform between point clouds A and B such that:

$$p^B = Hp^A$$

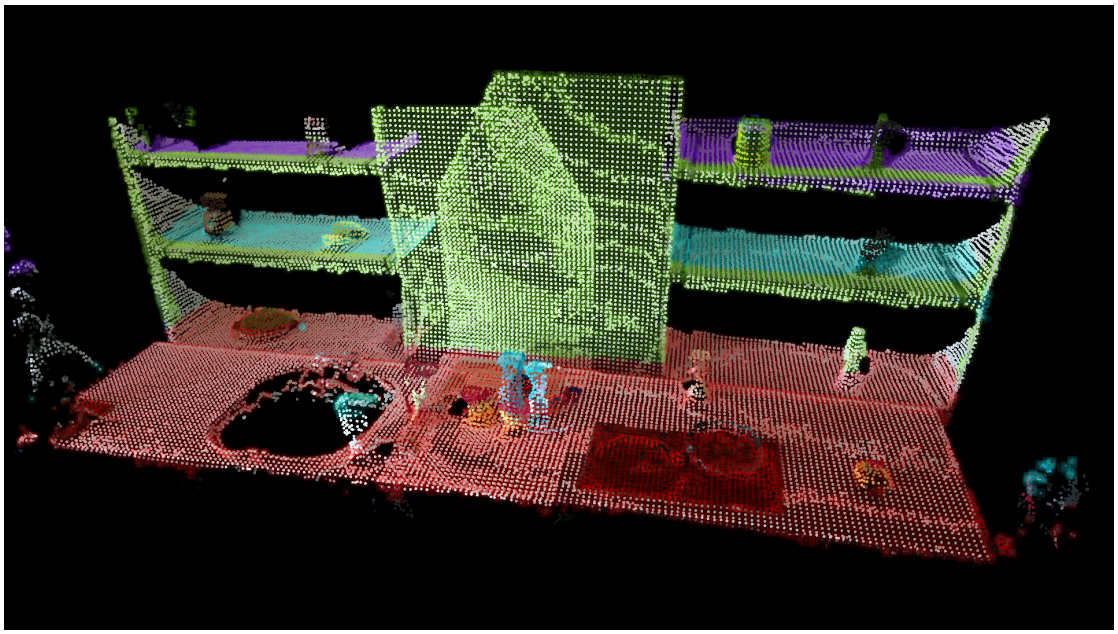

FIGURE 2.13: Planes matching result, $n = 3$.
In both cameras, matching planes are drawn with the same color.

We can define H' the transform that brings any plane equation in A, π^A, to its corresponding plane in point cloud B π^B. We can demonstrate () that:

$$H' = H^{-\top}$$

Finding H is then the same as finding H' that minimize plane distances:

$$\pi_i^B = H'\pi_i^A$$

If we write:

$$H' = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$

We then have for each matching planes:

$$\begin{cases} {n_i^B}^\top R = {n_i^A}^\top \\ {n_i^B}^\top t = \rho_i^B - \rho_i^A \end{cases}$$

By stacking all n vectors into a matrix N and all $\rho_i^B - \rho_i^A$ into a vector d, the problem became:

$$\begin{cases} N^B R = N^A \\ N^B t = d \end{cases}$$

We can then simply compute the least square solutions for any n-dimensional

distance metric where n is the number of matches. Especially, a useful metric could be for any n-d weight matrix W (most likely diagonal):

$$\mathcal{D}(x, y) = x^\top W y$$

This metric will allow to weight some planes differently according to the confidence (measured with point-plane distance standard deviation, total number of points etc.) we give to the precision of their plane detection, otherwise $W = I_n$ will be used. The final solution is:

$$\begin{cases} \tilde{t} = (N^{B\top} W N^B)^{-1} N^{B\top} W d \\ \tilde{R} = (N^{B\top} W N^B)^{-1} N^{B\top} W N^A \end{cases}$$

At the end we can then ensure the orthogonality of R using SVD decomposition. Finally we apply this rigid transform to the camera to create the final point cloud (fig. 2.14).

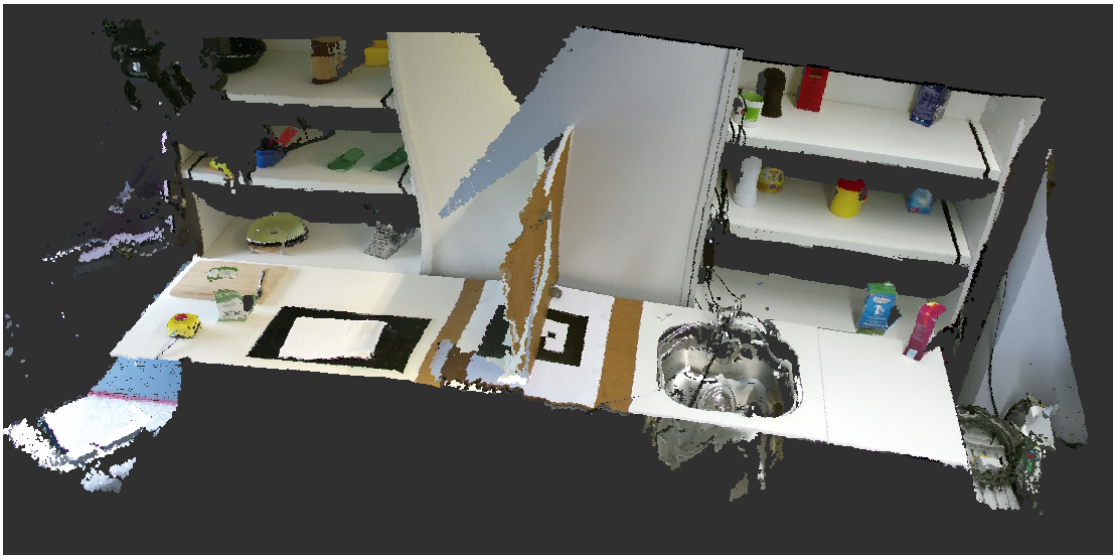

FIGURE 2.14: End result of plane matching with both cameras, $n = 4$.

To compute the rigid transform accurately with this technique it is needed to detect large planes inside point clouds. Hopefully it is really common in indoor environments to detect such primitives. However, this registration method requires 3 large non-degenerated (the more orthogonal the planes are, the better will be the result) pairs of planes which is a lot more restrictive, this is again a consequence of the lack of overlapping. To identify 3 planes means that the sensor is seeing

a corner but it is not probable to be able to see this corner in both views if the overlapping is small, moreover even if we do, the corner will be in the border of both view, meaning at least one plane will be small. In our setup, I chose to add manually a 3^{rd} (artificial) plane in the overlapping region to be able to run this algorithm (as seen in fig. 2.14). Even by choosing manually this plane it was not possible to place it such that both view contains a large part of this plane, the uncertainty of the computation is then decreased. Moreover the deformation of the point cloud in this region makes the y translation estimation inaccurate. To summarize, we can't expect to detect all 3 required planes. We may be able to use this algorithm in some specific cases but it requires to have knowledge of the scene and to know in advance that this scene fulfill these specific requirements which is not our case. A successful implementation of this algorithm has also been described in [Ara] and implemented in [Yua13] for SLAM mapping using mobile robot planar movement constraint.

Chapter 3

Implementation

All the above mentioned techniques appear to be efficient only under specific constrains. In our scenario none of them can be applied directly and we have to find a more suitable procedure for this project.

Even for humans, manually calibrating these point clouds is a hard task. Due to the lack of overlapping, it is difficult to notice whether a transform is wrong or not when focusing only on closest points matching (similarly to ICP algorithm). However, knowing that this is a scene containing large planes that can be seen in both views, we naturally tend to estimate the correctness of the registration by judging if the planes are aligned or not in the final result (this is similar to plane matching).

For this reason, it can be convenient to use simultaneously these two methods in our program to get the best result. Some similar approaches are described in the literature. The registration can be performed in one step, mixing keypoints (2D or 3D) and plane primitives as implemented in [DGFF13] and [TJRF13] or in 2 steps [PKC16], in this case keypoints are used as a robust method to register 2 similar point clouds while plane matching is used in a second registration process to improve point cloud alignment. This 2 steps methods can't be applied to low-overlapping point clouds because, as explained in chapter 2, both methods can't be applied separately in this case.

As we have seen, the most reliable information we have is given by planes that are visible from both cameras. The first step would then be to identify as many planes as possible. Depending on how many planes we can match, the remaining

degrees of freedom will be fixed with keypoints matching.
In this scene two mains plane can be identities, which means only one translation remains to be solved. Thus, only one point is needed, in theory, to solve the registration completely. The problem is that the point cloud quality doesn't let us find perfect matches. I will then try to identify as many matches as possible so that, on average, the error is lowered.

As explained previously, both registration using points matching or planes matching works in a similar way. It is still needed to adapt these equations to be able to find the transformation that minimize both points and planes distance in the same time.

3.1 Preprocessing

Before implementing any registration technique, few preprocessing steps are required to prepare data, fasten computation and improve

3.1.1 Downsampling

The first thing that appears is that point clouds are really heavy data structures and processing them will require a huge computation power. To be able to perform fast computation on point clouds I started by sub-sampling the point clouds to reduce the number of points.

Sub-sampling allow to significantly decrease the size of the data to process by picking equally spaced points which makes the final result keeping most of the relevant information of the point cloud.

The sub-sampling size can be chosen to adapt the precision/performance ratio (fig. 3.3).

3.1.2 Cutting

Another way to reduce the point cloud size is to cut it to remove non useful parts. This obviously requires to have some knowledge of the information received by the camera and will not be applicable in the case of a new unknown scene.

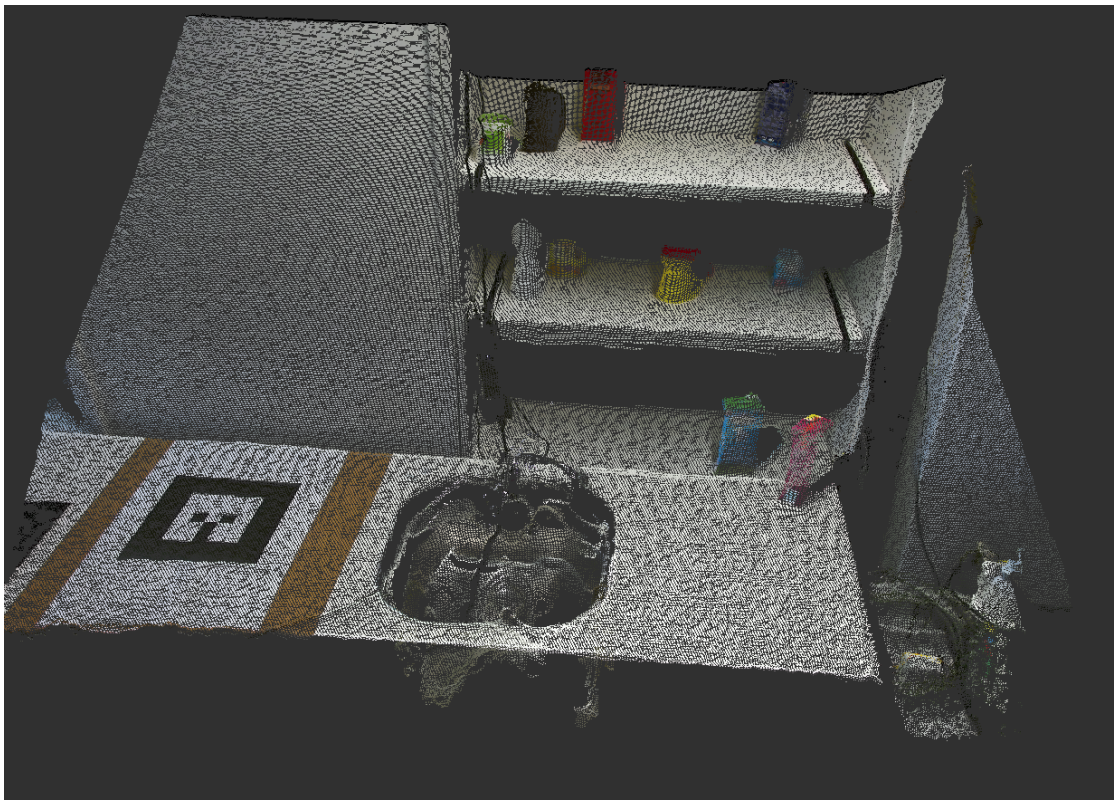

FIGURE 3.1: Full point cloud (16.588.800 points)

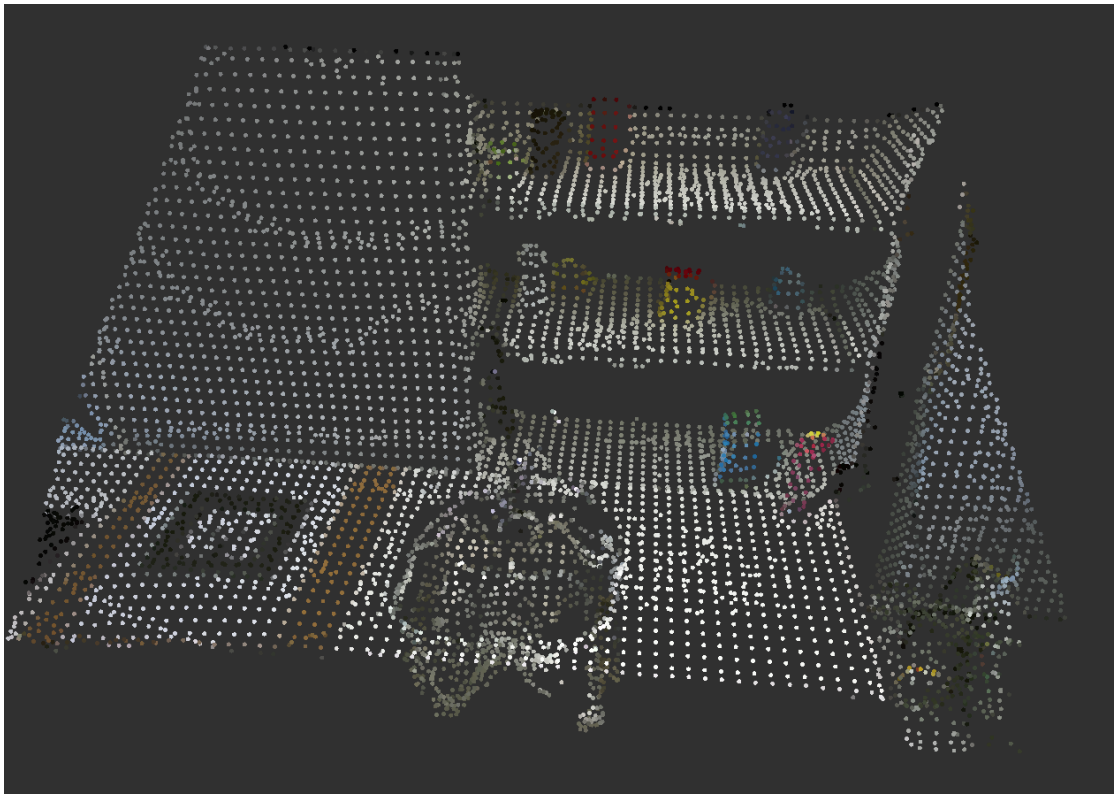

FIGURE 3.2: Sub-sampled point cloud (size: 3cm, 100k points)

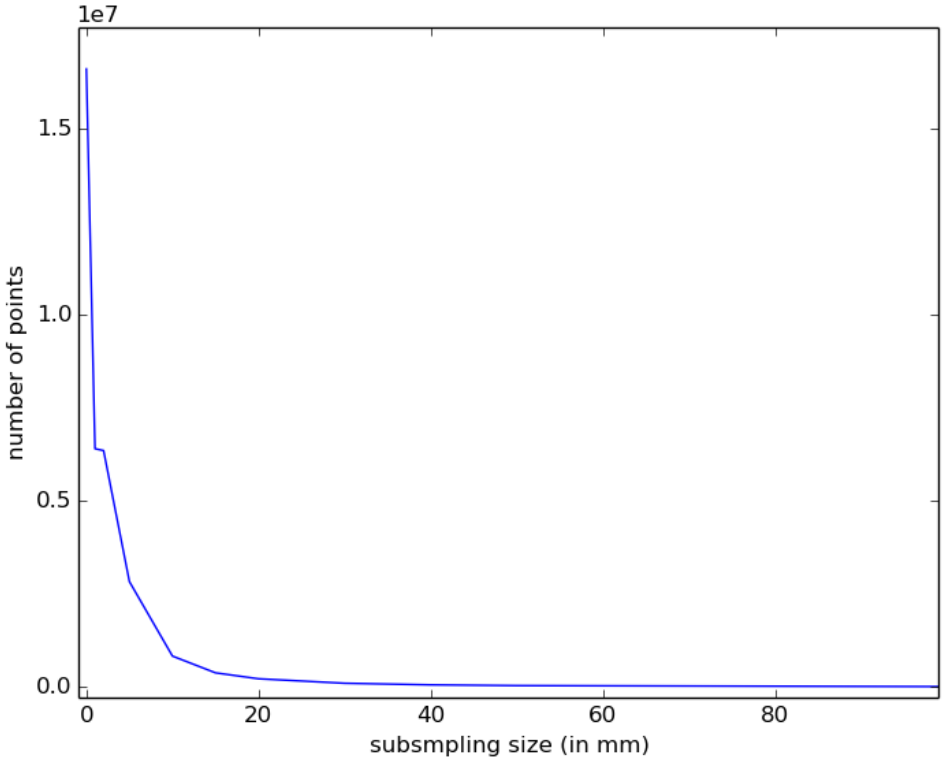

FIGURE 3.3: Example of evolution of the point cloud size against sub-sampling size.

However, cutting the point cloud will helps to quickly implement some registration techniques and will be used for testing and debugging purpose. In our project, when we try to fix only small camera displacements, we already have some rough knowledge on the scene. We can, for example, cut $z = 0$ points that corresponds to the floor, using some margin threshold we can reasonably except that the floor will be cuted without losing any points on the table ($z = 80cm$) unless the camera angle has been greatly modified.

In this example I cut the point cloud to keep only the points inside the working area.

3.1.3 Filtering

Finally, point cloud filtering is also a way I considered to decrease the size of the point cloud. I used radius filtering to delete outliers (points that don't have enough neighbors inside a chosen radius). However, this filtering was needing a

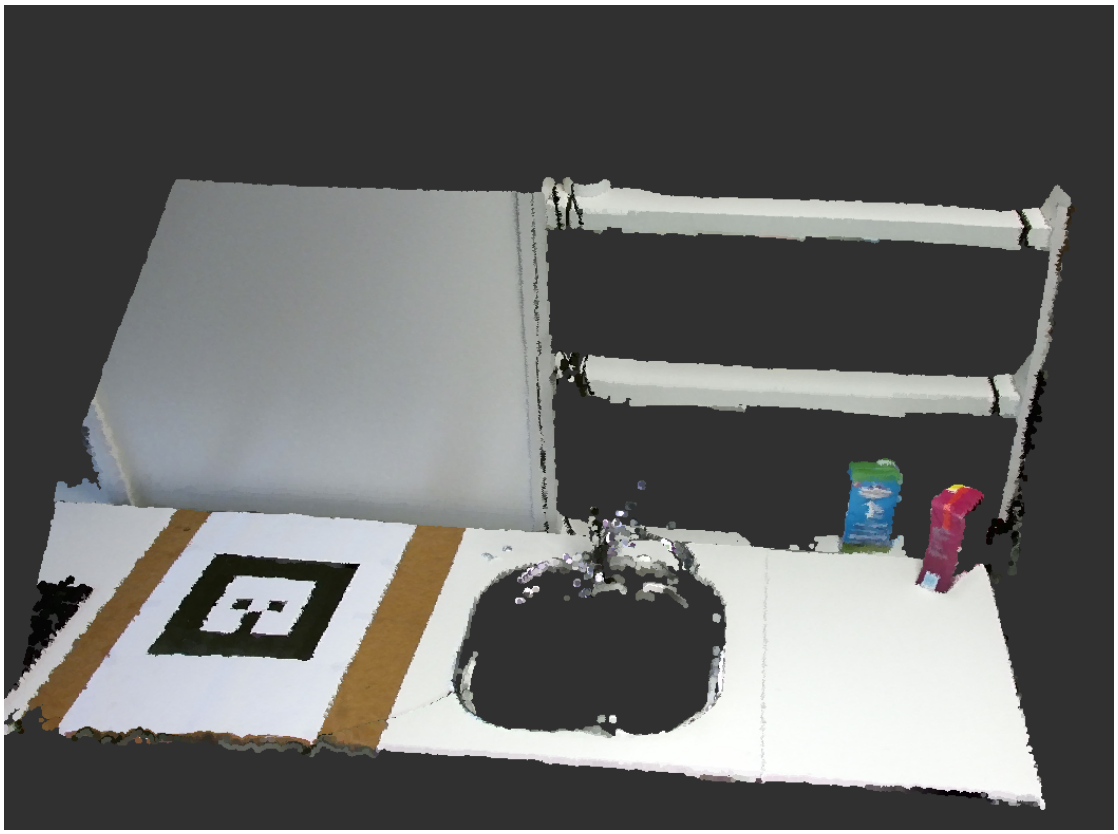

FIGURE 3.4: Example of point cloud cut using knowledge on the experiment environment.

lot of computation time compared to the small improvement it provides in this case. I no longer use this filtering in the preprocessing stage.

3.2 Plane Detection

The first processing step consists in extracting planes from the point clouds. I am using RANSAC (detailed in alg. 2) algorithm to detect the most relevant plane of the cloud. I used ?? library implementation of this algorithm to perform fast plane detection.

When we find a good model, if we note w the ratio between the number of inlying points and the total number of points, we can estimate the number of iterations needed to be confident enough that a correct model has been found. For example, if we iterate the algorithm k times, success probability p can be written:

$$(1 - w^3)^k = 1 - p$$

The minimum number of iterations is then:

$$k = \frac{log(1-p)}{log(1-w^3)}$$

In our setup, we can check how many iterations are needed. In the worst case, the first plane we want to detect is only 25% of the point cloud. It means that using 500 iterations, we have less than 0.1% chance not to find a good model.

When the first plane is extracted, we can continue by subtracting inliers from the point cloud and apply again the RANSAC algorithm, by iterating these steps I can extract as many planes as needed and they are extracted ordered by decreasing size. The idea is then to apply this plane detection on each point cloud to match these planes between both clouds.

3.3 Plane Matching

Plane matching consists in matching as many planes as possible between different cameras. These planes should be matched when they correspond to the same plane in the real scene. Based on [DGFF13], I implemented a plane color histogram matching as follow.
By extracting the color data from each point of point cloud subsets corresponding to a plane we can compute a color histogram representing color distribution in a given color space. This space can be RGB or HSV and can be computer for 3 1D color channels (fig. 3.5) or for a 3D color space (fig. 3.6).

Depending on the type of histogram different distance metrics can be used. As suggested in [DGFF13], overlapping areas can be computed for 1D histograms. I also performed K-means clustering on 3D histograms to determine the k=5 most common colors in each plane, it can be seen on fig. 3.6 that all colors are mostly grey which makes the matching difficult. It is then easy to compute a distance (Euclidian or Manhattan distance) between these centroids to find best matches. Given a plane color histogram from plane 3, I plot the distance to other planes color histograms captured from the 3 planes on both cameras on fig. 3.7.

However, none of these techniques achieves a robust matching due to the very similar colors in each plane and the very low number of common points shared by planes seen from different cameras.

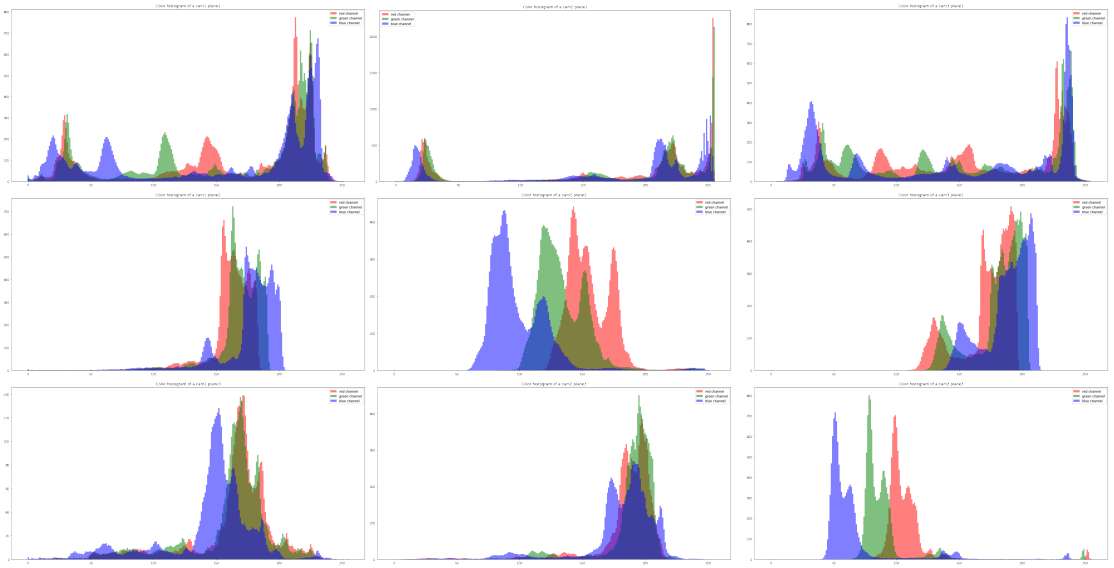

FIGURE 3.5: Planes RGB color histograms computed for 3 cameras on 3 different planes (table, wall and wooden plate), each line represents a plane, each column represents a camera.

As explained in section 2.4, I also tried to use PCL VFH global descriptor for matching planes. These techniques didn't allow me to match planes robustly neither. For this reason, I preferred to match planes by comparing normals, this technique doesn't rely on the later use of plane matches and is simple to implement. The only requirement is not to have a huge orientation difference between cameras. This means we need a prior knowledge of the rotation between cameras ($\pm 45°$). Given 2 planes $\pi_1 = (n_1^\top, -\rho_1)$ and π_2 which normal vectors n are chosen such that their product is positive, the metric f I use for matching is the following, with an arbitrary $\alpha = 1\, m^{-1}$:

$$f(\pi_1, \pi_2) = n_1 \cdot n_2 - \alpha \left| \rho_1 - \rho_2 \right|$$

The first term $n_1 \cdot n_2$ measures the cosine of normals angle between 0 (when they are perpendicular) and 1 (when they are parallel). The second term lowers this value if planes are far from each other. It allows planes to match with the right plane instead of another parallel one.

3.4 Keypoint Extraction

As explained in sec. 2.4, we usually don't detect 3 non-degenerated planes in the point cloud, especially when the overlapping is small. To solve this problem I am adding knowledge from the keypoints detection as done in 2.3. Method to estimate the transform from mixed data of planes and points is detailed in the next section.

Keypoints information is less precise than planes so we may want to give more importance to planes matching and use only keypoints to fix the last degree of freedom. We can define a weight between planes and points terms in the transform estimation equation. That way we give more importance to information provided by planes. However, we actually want to trust only planes for most of the degrees of freedom and use points matching only when it is needed. For example, we usually detect planes in 2 different orientations as we can see on fig. 2.13, which means we still have to fix one degree of freedom (translation). We want, in this case, to use keypoints only to determine this translation. To achieve this, I compute the intersection direction of planes and I project keypoints along this vector. That way, keypoints will only contribute to modify the translation part of the transform in this direction as shown on fig. 3.8. By doing this we ensure that keypoints matching doesn't introduce a bigger uncertainty for rotations and translations that are already accurately determined by planes.

3.5 Transform Estimation

My implementation of transform estimation using both matched points and planes is based on equations derived in [TJRF13]. As described in [Kho16], we can compute distances between planes (the value that should be minimized) in two different ways:

- Plane to Plane, which is an arithmetic distance obtained by comparing plane equations. It is composed by the vector distance between plane normals and the scalar distance between ρ parameters.
- Point to Plane, a geometric distance obtained by averaging point to plane distances from one plane to another.

I am using the plane to plane distance since it is by far the fastest to compute. My implementation is then based on these equations derived in [Kho16]. Equations are similar to the ones detailed in sec. 2.3.5 but we are adding terms (in red) related to plane normals. Resulting equations are then a generalized form of the equations from sec. 2.3.5 and 2.4:

$$R = U \begin{pmatrix} 1 & & \\ & 1 & \\ & & det(UV^\top)) \end{pmatrix} V^\top$$

Where U and V are computed from the SVD of the 3×3 correlation matrix K:

$$K = UDV^\top = \sum_i p_i p_i^{s\top} + \sum_j w_j n_j^t n_j^{s\top}$$

Where w_j is the weight associated to planes with normals n_j as defined in sec. 2.4 when W is diagonal.

To compute the translation part we have to define A and b:

$$A = \begin{pmatrix} MI_3 \\ w_1 n_1^{t\top} \\ \vdots \\ w_N n_N^{t\top} \end{pmatrix}$$

$$b = \begin{pmatrix} M(p'^t - Rp'^s) \\ w_1(\rho_1^s - \rho_1^t) \\ \vdots \\ w_1(\rho_N^s - \rho_N^t) \end{pmatrix}$$

$$t = (A^\top A)^{-1} A^\top b$$

3.6 Program Description

All these processing steps are run in separate ROS nodes and interact as explained on fig. 3.9. Let's also describe briefly the structure of the program developed for this project.

3.6.1 Classes

- Cloud

  This class is used to convert Point Cloud data types from/to sensor_msgs and PCL types. It also allows to subscribe and publish topics and to move the point cloud from the User Interface.

- Preprocessing

  Preprocessing object is used to perform preprocessing steps; Point Cloud cutting and subsampling and set up the User Interface allowing the user to manage parameters.

- Plane Detector

  This object is simply extracting planes using RANSAC algorithm and provides the graphical interface to tunes parameters.

- Matching

  The Matching class is performing every process linked to keypoints and planes matching. It extracts objects on the table, search for keypoints and compute their descriptor representation, find the matching between them and the matching between planes. Finally it allows parameter tuning from the User Interface.

- TransformEstimator

  This class correspond to the last step of the pipeline, it subscribe to planes and keypoints matching topics and use them to estimate the transform between point clouds.

3.6.2 Nodes

- icp_node

  This node subscribe to 2 Point Clouds topics, synchronize them and apply ICP algorithm to estimate their relative position.

- merging_node

  Used to merge 2 Point Cloud topics and merge them into one full cloud. It also provide to the user some parameters to cut the overlapping region through the User Interface.

- tf_node

  This node filter a moving TF. It is used to average or filter the result of transform estimation.

- registration_node

  Main node that call instances of Cloud and Merging classes.

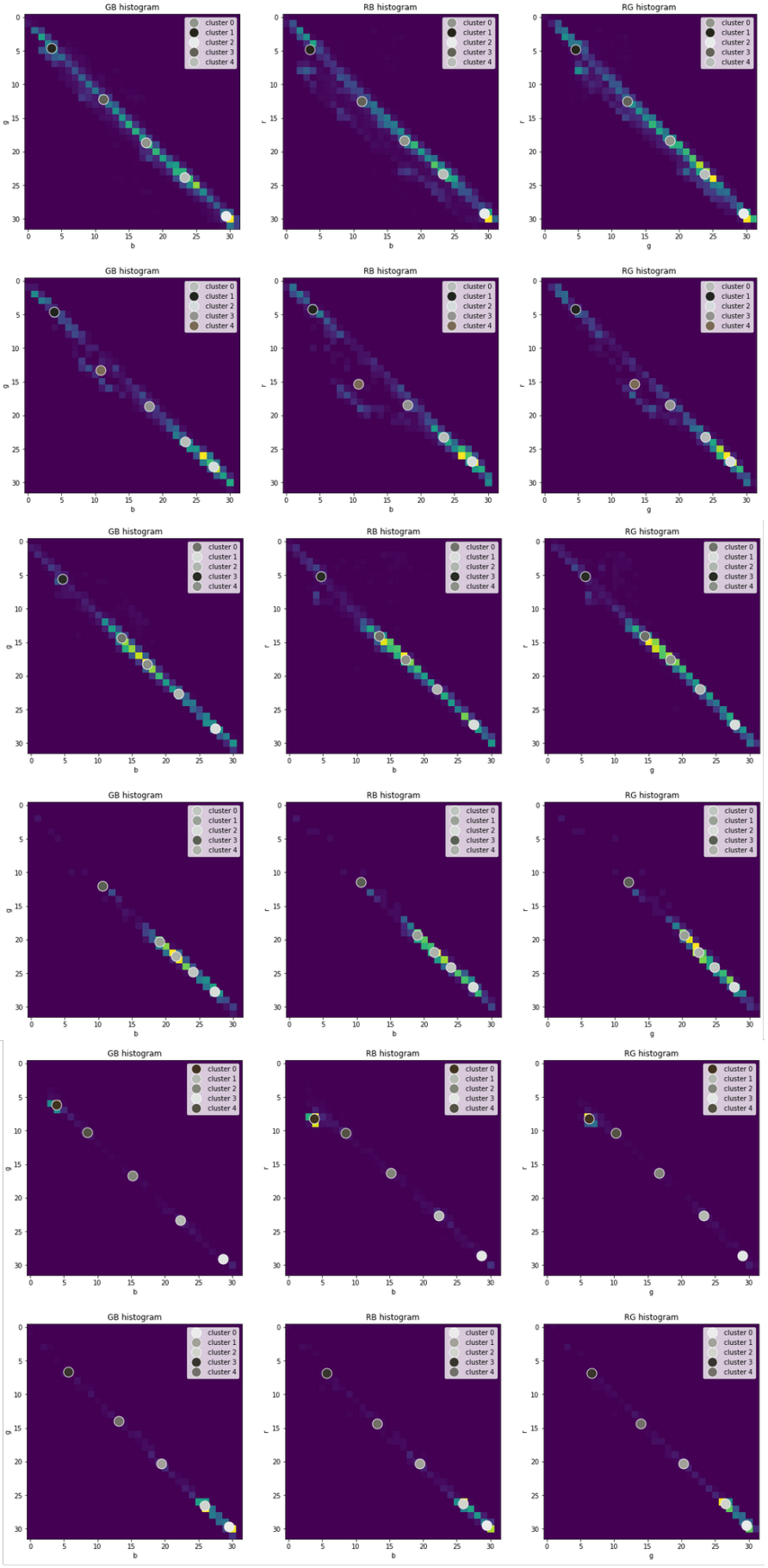

FIGURE 3.6: 2D projections of 3D color histograms for each planes. Lines correspond to plane 1 seen from cam1 (line 1) and from cam2 (line 2) then plane 2 (lines 3, 4) and plane 3 (lines 5, 6). Centroids of K-mean clustering (k=5) are shown with the corresponding color.

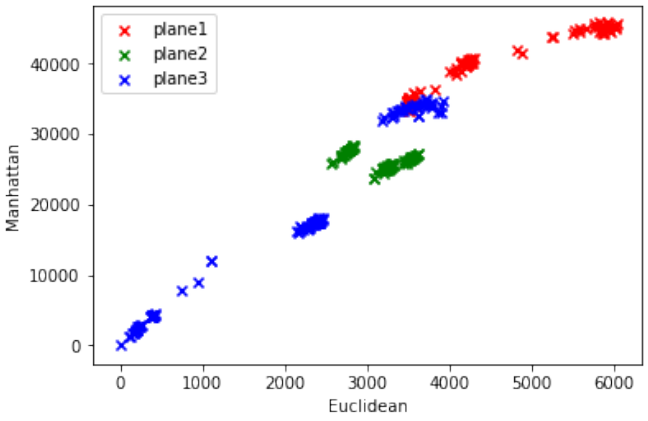

FIGURE 3.7: Distance between K-Means centroids computed between planes color histograms from different cameras and time and a color histogram from plane 3 seen from cam 1.

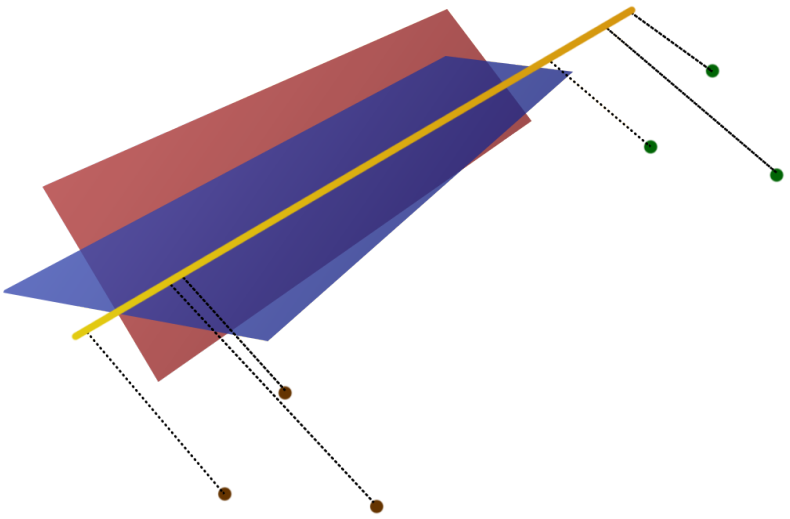

FIGURE 3.8: Projection of keypoints on plane intersection axis.

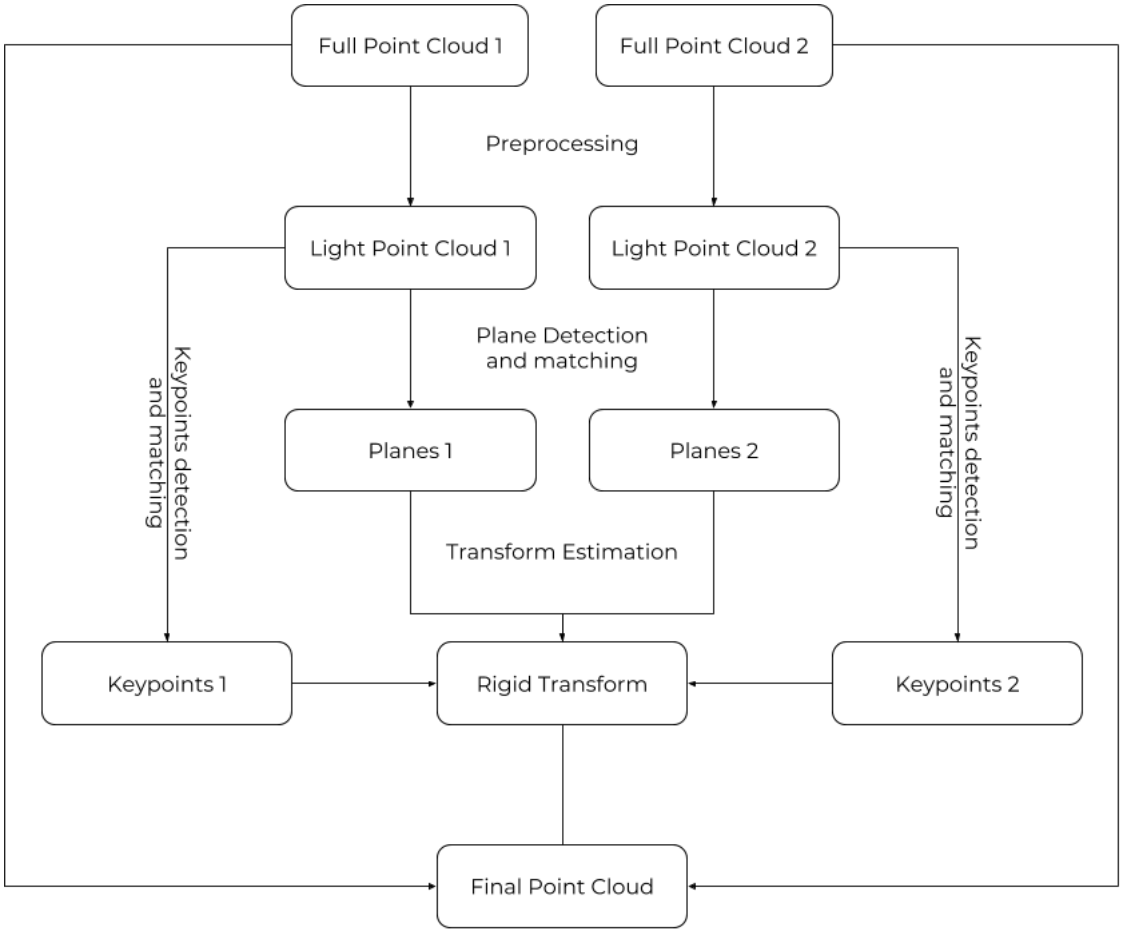

FIGURE 3.9: Registration pipeline.

Chapter 4

Methodology

4.1 Environment

As explained previously (1.2), the purpose of this work is to compare registration techniques in a specific case where point cloud registration is challenging. I realized a benchmark of methods described in the section 2 and the modified one detailed in section 3. Each method is compared in different scenarios ; registering point clouds acquired from our real sensors in the kitchen scene, performing the registration on the kitchen with simulation data and finally with common point cloud databases to compare with the results from literature.

4.1.1 Real scenario

Our real scenario is composed by 2 kinects filming a kitchen counter top and shelves filled with common kitchen objects.

In this scenario on e of the main point is to define accurately the ground truth, is is possible to measure with few mm of error the translation between cameras but their relative orientation is much more difficult to measure. To evaluate this ground truth as accurately as possible I used all the knowledge we have: defining the initial guess as the manually measured translation and assuming equal orientation (cameras are facing in the same direction). I then align point clouds more accurately with the plane detection method and finely adjust the transform by hand so that point clouds match exactly.

4.1.2 Simulation

Using 3D models of kitchen furnitures, I recreated the scene inside gazebo simulator which is compatible with ROS and I was able to publish point clouds topic using simulated kinects sensors. This setup is then working exactly in the same way as the previous one, our program can subscribe and share point cloud topics through ROS.

In this case the ground truth is perfectly know as it is defined inside the gazebo world model where I placed the sensors with absolute coordinates and orientation.

4.2 Variables

The objective of the algorithm is to estimate the transform between two kinects, I then need to compare this transform (rotation and translation) with the ground truth. It is difficult to find a meaningful scalar metric to compare transform matrices. We can nevertheless compare separately translation and rotation parts, by computing the overall translation distance l and rotation angle θ around the fix axis a. Let $\mathcal{T}$ be the 4×4 homogeneous matrix with rotation part described by coordinates of basis vectors u, v, w the target frame written in the source frame and t the translation vector between both frames' origins (fig. 4.1).

$$\begin{bmatrix} u_x & v_x & w_x & t_x \\ u_y & v_y & w_y & t_y \\ u_z & v_z & w_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We'll describe this transform with the following two metrics [GP00].

$$\left.\begin{aligned} l(\mathcal{T}) &= \left\| t \right\| \\ &= \sqrt{t_x^2 + t_y^2 + t_z^2} \end{aligned}\right\} \quad \text{the translation metric} \tag{4.1}$$

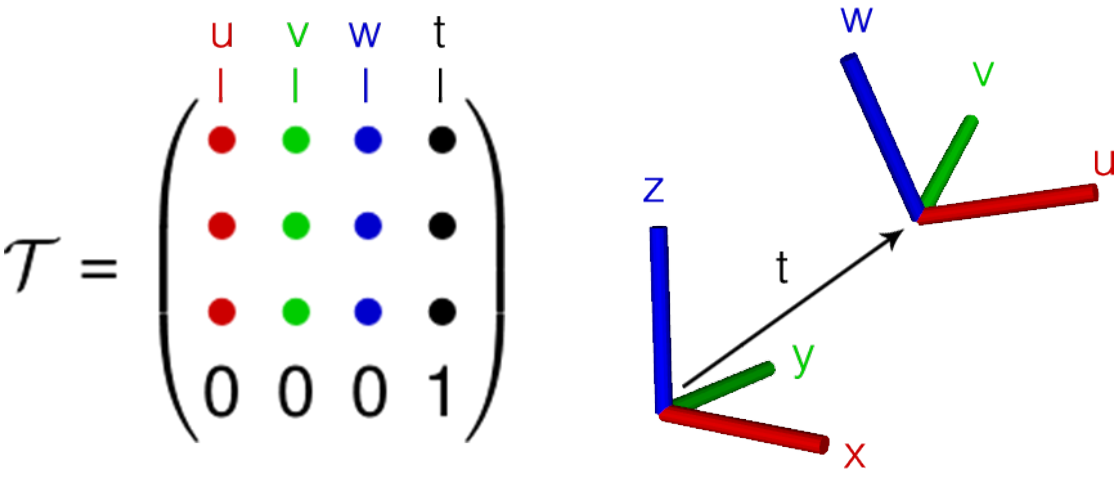

FIGURE 4.1: Transform matrix description.
$\mathcal{T}$ describes the transform between xyz and uvw.

Using the fact that the rotation axis is

$$\mathbf{a} = \begin{pmatrix} w_y - v_z \\ u_z - w_x \\ v_x - u_y \end{pmatrix}^T$$

with

$$\left\|\mathbf{a}\right\| = 2sin(\theta)$$

and

$$Tr(R) = 1 + 2\cos\theta$$

We can compute the rotation angle from the matrix coefficients:

$$\left.\theta(\mathcal{T}) = \arctan\left(\frac{\sqrt{(w_y - v_z)^2 + (u_z - w_x)^2 + (v_x - u_y)^2}}{u_x + v_y + w_z - 1}\right)\right\} \quad \text{the rotation metric} \tag{4.2}$$

Chapter 5

Results

Each registration method is applied to estimate cameras relative positions on a recorded sequence containing 300s records for both kinect sensors. ICP method is applied in 3 different ways:

- *ICP* is the classical way of applying ICP for registration, we try to match one cloud with the other
- In *ICP_PointCloud* I apply the ICP for each point cloud to match with a given cloud of the complete scene. This cloud is a reconstruction previously prepared from both views, this techniques obviously requires a prior knowledge but it is good as a comparison with other methods. This point cloud contains objects on the table which may fail the ICP as it would try to match different shapes form the live point clouds.
- *ICP_CAD_model* is the same method than the previous one but using a point cloud generated from a CAD model of the scene instead of a real reconstruction of the scene. This model doesn't contain any object that could help registration (especially the y translation).

Only the first method is satisfying our requirements for this work (no previous knowledge), the 2 others are applied for comparison purpose but assume that we have already solved the registration problem (reconstructed point cloud) or that we have a perfect knowledge of the expected result (CAD model).

Other methods are the ones explained in the previous sections:

- *plane_matching*, registration using only plane matching, thus a 3^{rd} plane is added as explained in section 2.4
- *keypoints_matching*, using only keypoint matching from section 2.3
- *my_method*, the method detailed in section 3, mixing both previous techniques.

5.1 Distance and Angle Error

From the estimated transform $\mathcal{T}_{1\longrightarrow 2}$ I extract translation vector (t_x, t_y, t_z) and Euler angles (r_x, r_y, r_z) as well as the 2 metrics explained in the previous section (l and θ). This 2 metrics give a easy overview of the transformation error in both translation and rotation in order to compare each method but the 6 other values will be used for a more detailed understanding of the errors.

5.1.1 Real Scenario with Bad Initial Guess

For this first experiment I set a bad initial guess as if we are trying to calibrate the perception system for the first time. Knowing the ground truth, this initial guess is chosen such that the transform between both point clouds is 30° of rotation on each axis and a 50cm translation.

First, we can observe that in our setup, matching methods (planes and/or keypoints matching) still gives good results while ICP methods doesn't converge to a satisfying result since they are really sensible to modification on the starting position. Thus, no comparison can be made with ICP methods because they are providing satisfying results. We can verify that plane matching is performing better for rotation estimation as explained in section 3.3 while keypoints matching is less effective for rotation estimation but gives a good translation estimation. The mixed method is keeping relevant from both methods and provide a good estimation for both translation and rotation with a small deviation which makes this method more reliable than the 2 others.

We can notice that even for the mixed method, the standard deviation is larger for the Y translation estimation. This was expected due to the symmetry of the scene. This translation is determined only from keypoint matching and

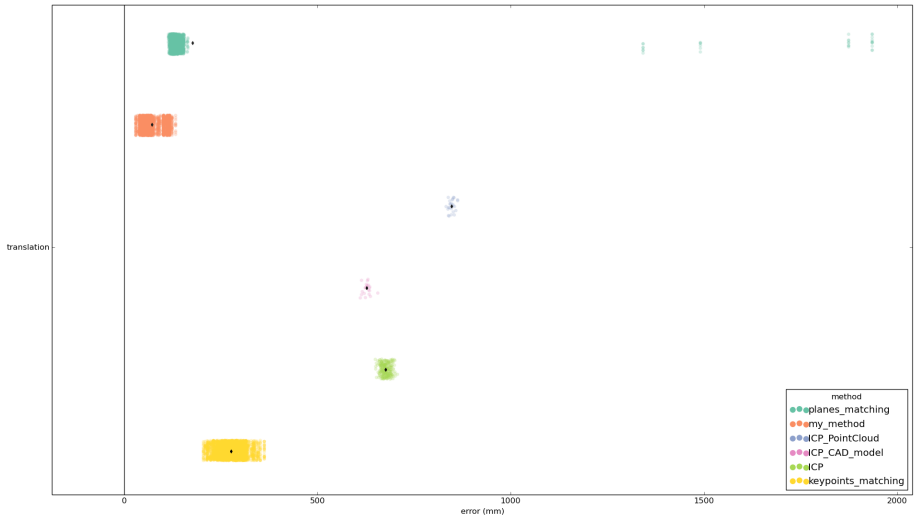

FIGURE 5.1: Comparison of overall translation error for each method.

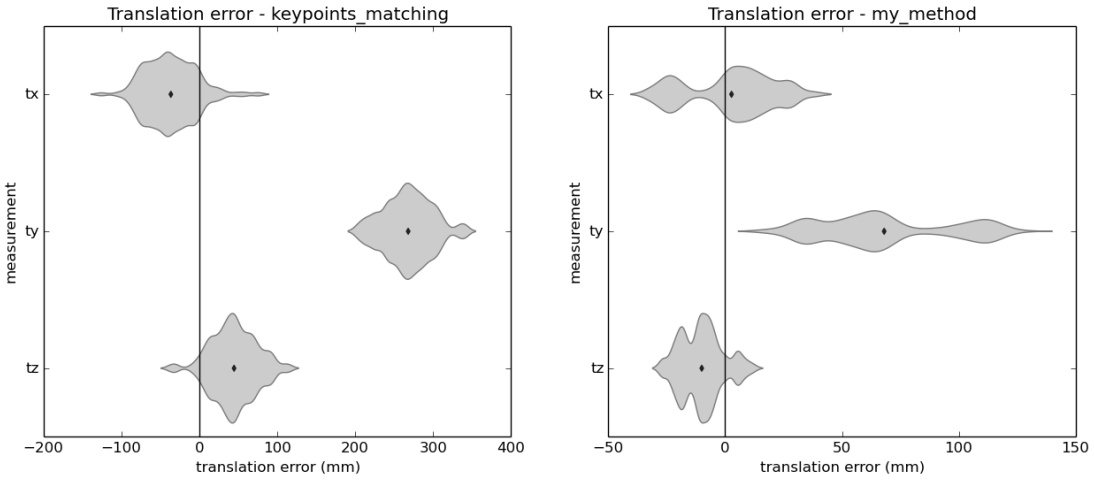

FIGURE 5.2: Detailed translation error distribution.

not from plane matching as other variables. Plane matching provides a precise translation estimation in the normal direction because it uses all the plane points position to align both clouds. Keypoints matching consists in matching few points that are determined with a large uncertainty.

However, as explained earlier, we can reduce the effect of these deviation by averaging the transform, especially during the first calibration process.

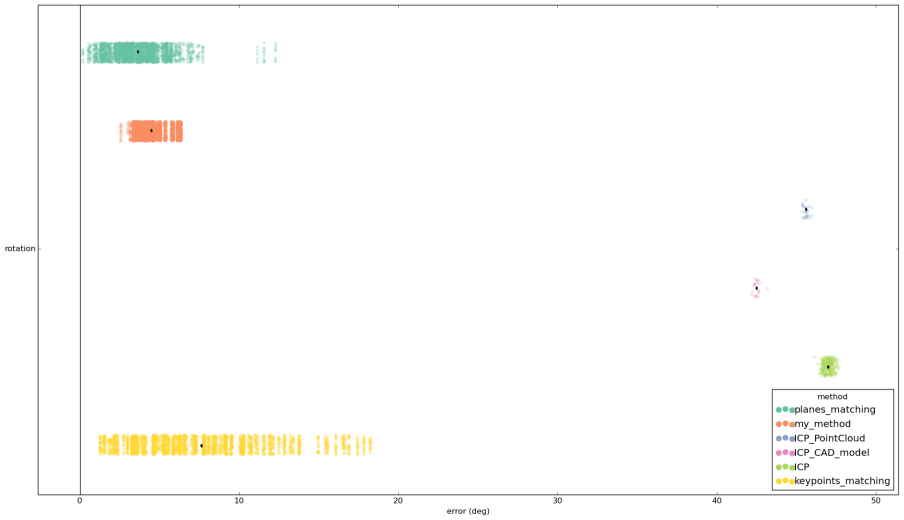

FIGURE 5.3: Comparison of overall rotation error for each method.

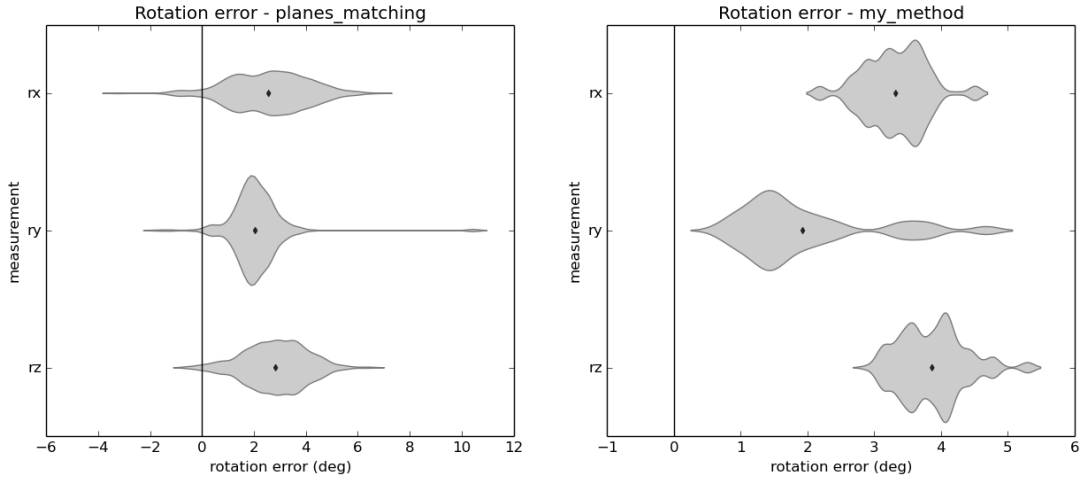

FIGURE 5.4: Detailed rotation error distribution.

5.1.2 Real Scenario with Small Initial Error

In this experiment I start with an initial guess much smaller than in the previous one ; only 10° of rotation on each axis and 20cm of translation. Even with this initial pose, the direct ICP method was not able to converge reliably to the solution, I was needed to tune really precisely the algorithm parameters for each attempt. I then choose to focus on matching methods since we are trying to compare automatic registration method and ICP registration can't be considered as an automatic method in this particular case.

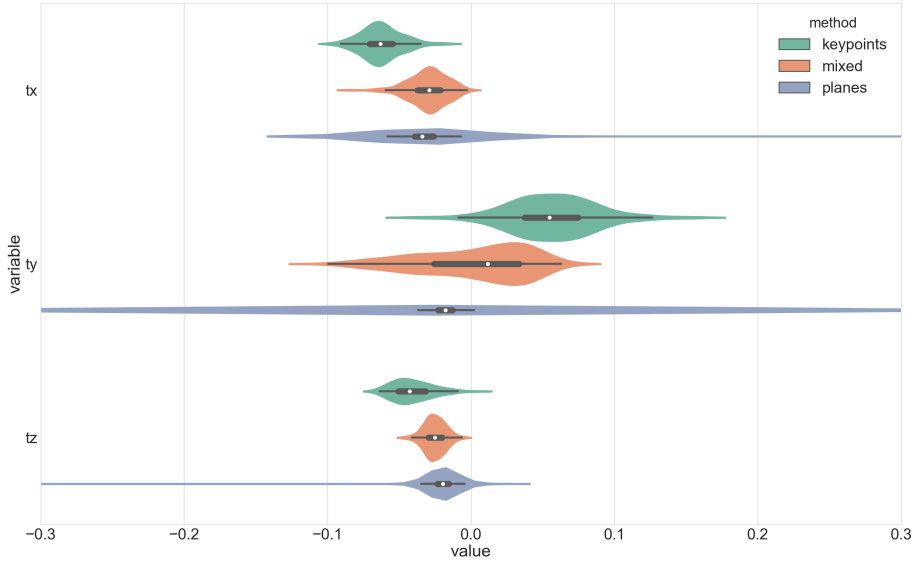

FIGURE 5.5: Translation error distribution in m.
White dot, thick and thin black lines correspond respectively to the median, interquartile range and 95% confidence interval.

In this experiment with a smaller initial error, our 3 matching methods behaviour is mostly unchanged. Mean values and standard deviation are detailed in table 5.1.

First, we notice that planes matching method obtains values with a very large deviation in the translation estimation due to outlying values. These outliers can be caused by wrong plane matching or by an error in the plane detection.

For each method, the Y translation is also more spread out than the 2 other translations as we noticed in the first experiment. In the case of plane matching ty uncertainty is very large because the 3^{rd} plane used to determine this translation is very deformed because it is placed in the overlapping region.

Keypoints matching method is, in contrary, less effective for rotation estimation, this can be understood as this method uses a matching of points only located in the center of the scene to align point clouds while methods using planes align planes that are defined by a large number of points spread out on the entire scene. Once again, the rotation estimation is slightly better with the mixed method than the plane matching.

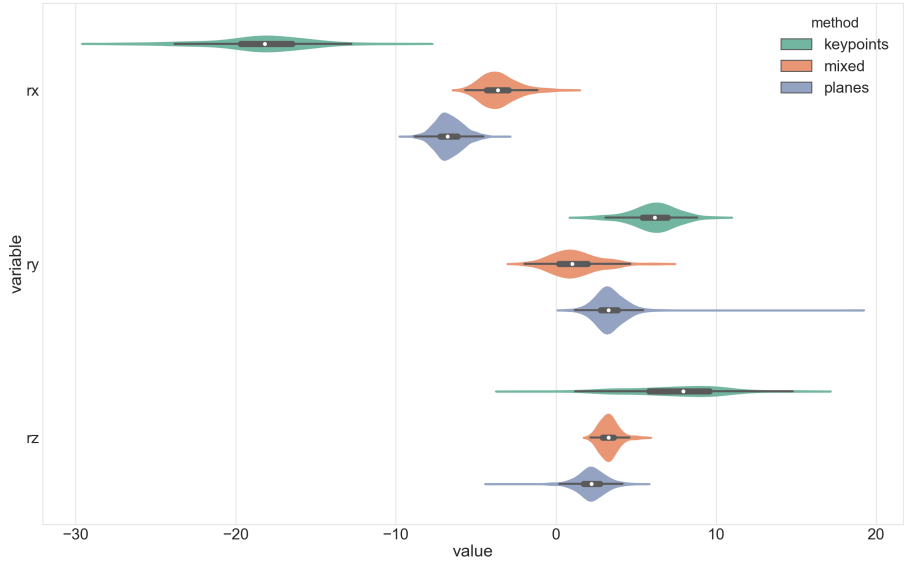

FIGURE 5.6: Rotation error distribution in degrees.

Final registration result with the mixed method is presented in fig. 5.7.

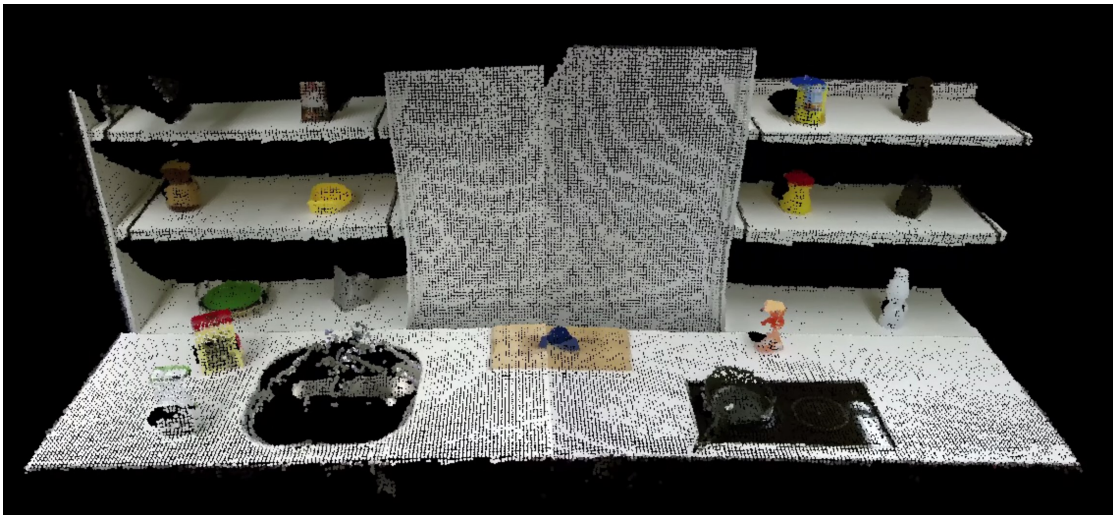

FIGURE 5.7: Final registration result using mixed method.

5.2 Computing Speed

We can evaluate algorithm's speed by comparing the number of transform estimations computed during the experiment and deduce the average number of frames (or estimations) per second.

The fastest algorithm is by far plane matching registration because it only involves RANSAC planes detection which doesn't need a lot of computation. In contrary the keypoint detection, feature estimation and matching used in other methods is needing a lot more computation. The reason why the mixed methods is faster than keypoints matching only is that as it is based mainly on plane detection, only few keypoints are needed and only objects in the overlapping region are considered while keypoints matching needs to detect keypoints also on the table and wall to be accurate which leads to the need of processing a lot more keypoints.

Method	Variable	Mean Value	Standard Deviation
Keypoints Matching	tx	-0.0625	0.0138
	ty	0.0569	0.0300
	tz	-0.0409	0.0136
	rx	-18.4	2.89
	ry	6.13	1.38
	rz	7.49	2.86
Planes Matching	tx	-0.0275	0.138
	ty	-0.0410	0.620
	tz	-0.0203	0.0331
	rx	-6.69	0.842
	ry	3.36	1.04
	rz	2.22	0.891
Mixed Method	tx	-0.0300	0.0132
	ty	0.00243	0.0374
	tz	-0.0252	0.00734
	rx	-3.55	1.11
	ry	1.20	1.45
	rz	3.30	0.576

TABLE 5.1: Detailed mean and standard deviation for each variable. Translation in m and rotation in degrees.

Method	FPS
Keypoints	0.84
Planes	7.58
Mixed	1.59

TABLE 5.2: Computing speed during the first experiment (in FPS).

Chapter 6

Conclusion

According to our results, this method mixing planes and keypoints matching leads to better results in this project. Indeed this method obtains the smallest error in translation over all the methods tested. For the rotation part, plane matching is performing as good as the mixed method in average but has a larger standard deviation. From these measures we can clearly see that matching methods (keypoints, planes and mixed) perform a lot better on this registration problem than ICP registration that would require a lot more overlapping.

For our perception system which aims to provide a real-time reconstruction of the scene, this implemented solution can be used to obtain a real-time reconstruction of the scene with few centimeters of error. By averaging the estimations on few seconds we can obtain a more accurate reconstruction but the framerate would be reduced. By only using time filtering we can change the balance between speed and accuracy to adapt to our need.

This algorithm can then be used in this project to calibrate camera for the first time, to fix calibration errors when needed and even track the position of a moving camera in real-time if the system can deal with few centimeters of error.

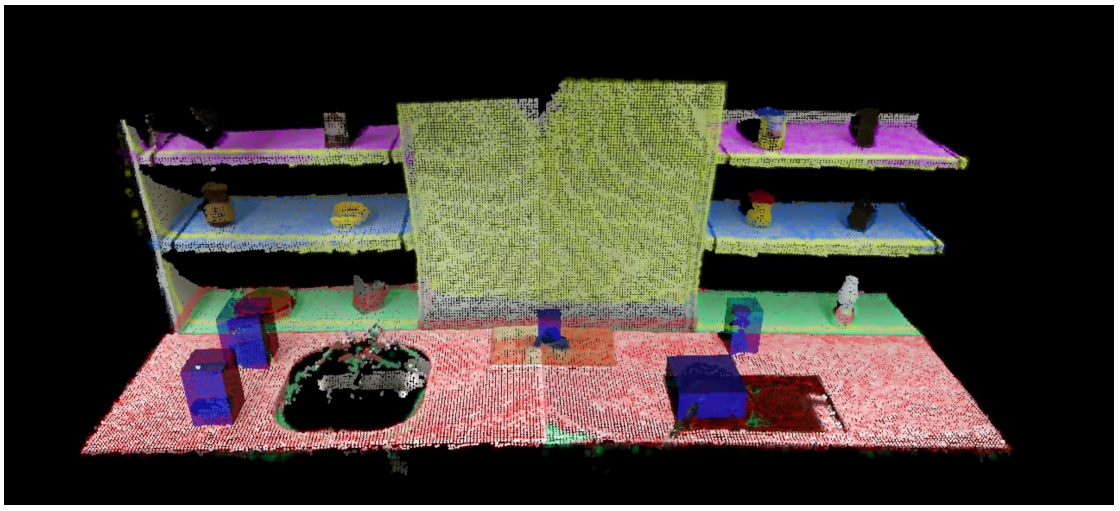

FIGURE 6.1: Final result with segmented planes and objects obtained by running perception program with our provided 3D scene.

Chapter 7

Future Work

7.1 Plane Matching

We have seen that the plane matching techniques I used were not demonstrating robustness and the final method I used need to assume that clouds are already barely aligned at the beginning. Finding a robust plane matching technique would then extend the range of use of this program. Another way to solve this problem could be to apply a first alignment method that roughly register point clouds as done in [PKC16] but with low overlapping this appear to be very hard using only 3D keypoint matching.

7.2 Using RGB data

This work focus on using only the point clouds data, however the camera also provides high definition images that can be used to find 2D keypoints with a better precision. By using depth sensor data to estimate the 3D pose of these points we may be able to improve greatly the point matching results.

7.3 Improve tracking

In this project we use the registration process to calibrate fixed cameras. If we want to apply this algorithm for camera position tracking, some additional work

can be done to improve the result such as implementing camera pose prediction from past estimations to smooth tracking errors.

7.4 GPU implementation

All this work has been implemented for CPU computation but, by installing GPU version of libraries and adapting the code to use GPU version of some functions we can improve the computing speed of this algorithm by accelerating keypoints detection, feature extraction and matching, plane detection or ICP registration.

Acronyms

AR Augmented Reality. 5

CAD Computer-Aided Design. 38

CSHOT Color SHOT. 12

FPS frame per second. v, 46, 57

GPU Graphics Processing Unit. v

HSV Hue, Saturation, Value. 26

ICP Iterative Closest Point. 6–10, 21, 31, 38, 39, 41, 47, 55

IR Infra-Red light. 3

ISS Intrinsic Shape Signature. 11, 12, 14, 55

MSE Mean Square Error. 7

NARF Normal Aligned Radial Feature. 11

OS Operating System. v

PC Point Cloud. v

PCL Point Cloud Library. v, 9, 12, 16, 27, 30

RANSAC RANdom SAmple Consensus. 15, 25, 26, 30, 44

RGB Red Green Blue. v, 5, 26

ROS Robot Operating System. v, 36, 53

SHOT Signature of Histograms of OrienTations. 51

SIFT Scale-invariant feature transform. 11, 55

SLAM Simultaneous Localization And Mapping. 20

SVD Singular Value Decomposition. 14, 19, 29

UI User Interface. 30, 31

VFH Viewpoint Feature Histogram. 16, 27

Glossary

extrinsic calibration Calibration of the camera that consists in finding the position of the camera in its environment. 53

intrinsic calibration Calibration of the camera that consists in finding parameters that depends only on the device like focal length, focal center, deformation matrix etc. This calibration is required for extrinsic calibration. 12

library Collection of resources used by computer programs to gain behaviors implemented inside that library with to implement that behavior itself. v, 13, 16, 25

middleware Computer software that provides services to software applications beyond those available from the operating system. v

open source Software which source code is free to access and redistribute to encourage peer production and knowledge sharing. v

ROS node In ROS framework, a node is a process that perform some computation and communicate with other nodes. A program is composed by a combination of nodes. 30

TF ROS object representing a Rigid Transform between 2 frames. Given a world reference frame, a TF can then be used to describe a robot joint frame. In this work it is mainly used to represent the transform between point clouds.. 31

topic In ROS framework, a topic is a channel that allows nodes to exchange messages by subscribing or publishing data from and to other nodes.. 30, 31

voxel 3D equivalent of pixel, there are used to respresent a 3D map as a grid of elementary cubes.. 2

List of Figures

List of Tables

Bibliography

[Ara] M. Aravindh.
kinect registration.
http://asimov.usc.edu/~aravindh_m/.

[DGFF13] Mingsong Dou, Li Guan, Jan-Michael Frahm, and Henry Fuchs.
exploring high-level plane primitives for indoor 3d reconstruction with a hand-held rgb-d camera. In Jong-Il Park and Junmo Kim, editors, *Computer Vision - ACCV 2012 Workshops*, pages 94–108, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GP00] Major F Gendron P., Lemieux S.
quantitative analysis of nucleic acid three-dimensional structures.
http://www.major.iric.ca/wiki/index.php/Homogeneous_Transformation_Matrix_Distance_Metric, 2000.

[Kho16] Kourosh Khoshelham.
closed-form solutions for estimating a rigid motion from plane correspondences extracted from point clouds.
ISPRS Journal of Photogrammetry and Remote Sensing, 114:78 – 91, 2016.

[PKC16] Yong Kwon Cho Pileun Kim and Jingdao Chen.
target-free automatic registration of point clouds, 2016.

[SHR17] Olga Sorkine-Hornung and Michael Rabinovich.
least-squares rigid motion using svd, 2017.

[TJRF13] Y. Taguchi, Y. Jian, S. Ramalingam, and C. Feng.
point-plane slam for hand-held 3d sensors. In *2013 IEEE International Conference on Robotics and Automation*, pages 5182–5189, May 2013.

[Tom13] Federico Tombari.
PCL keypoints and features.
http://www.pointclouds.org/assets/uploads/cglibs13_features.pdf, 2013.

[TSS11] F. Tombari, S. Salti, and L. Di Stefano.
a combined texture-shape descriptor for enhanced 3d feature matching. In *2011 18th IEEE International Conference on Image Processing*, pages 809–812, Sept 2011.

[Yua13] Zehui Yuan.
Plane-based 3D Mapping for Structured Indoor Environment. PhD thesis, Politecnico di Torino, february 2013.